

# Przetwarzanie danych w dużej skali

---

NIEZAWODNOŚĆ, SKALOWALNOŚĆ  
I ŁATWOŚĆ KONSERWACJI SYSTEMÓW



Tytuł oryginału: Designing Data-Intensive Applications:  
The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-4066-4

© 2018 Helion S.A.

Authorized Polish translation of the English edition of *Designing Data-Intensive Applications* ISBN 9781449373320 © 2017 Martin Kleppmann.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[http://helion.pl/user/opinie/przdan\\_ebook](http://helion.pl/user/opinie/przdan_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

*Technologia jest potężną siłą w naszym społeczeństwie. Dane, oprogramowanie i komunikację można wykorzystać w złym celu — do umacniania niesprawiedliwych struktur władzy, do łamania praw człowieka i do ochrony własnych interesów. Ale można też posłużyć się nimi, aby czynić dobro — przekazywać głos mniejszości, otwierać możliwości dla każdego i zapobiegać katastrofom. Tę książkę dedykuję wszystkim, którzy działają na rzecz dobra.*





*Informatyka to kultura popularna. [...] Kultura popularna pogardza historią. W kulturze popularnej liczy się tylko indywidualność i poczucie uczestnictwa. Nie ma ona nic wspólnego ze współpracą, przeszłością lub przyszłością — ważne jest życie tu i teraz. Uważam, że to samo dotyczy większości osób piszących kod dla pieniędzy. Nie mają oni pojęcia [skąd wzięła się ich kultura].*

— Alan Kay, z wywiadu dla magazynu „Dr Dobb’s Journal” (2012)



<b>Przedmowa .....</b>	<b>11</b>
<b>I Podstawy systemów danych .....</b>	<b>17</b>
<b>1. Niezawodne, skalowalne i łatwe w konserwacji aplikacje .....</b>	<b>19</b>
Myślenie o systemach danych .....	20
Niezawodność .....	22
Skalowalność .....	26
Łatwość konserwacji .....	33
Podsumowanie .....	36
<b>2. Modele danych i języki zapytań .....</b>	<b>41</b>
Model relacyjny a model oparty na dokumentach .....	42
Język zapytań o dane .....	54
Modele danych przypominające graf .....	60
Podsumowanie .....	72
<b>3. Przechowywanie i pobieranie danych .....</b>	<b>79</b>
Struktury danych używane w bazie .....	79
Przetwarzanie transakcji czy analityka? .....	98
Bazy kolumnowe .....	103
Podsumowanie .....	110
<b>4. Kodowanie i zmiany .....</b>	<b>119</b>
Formaty kodowania danych .....	120
Sposoby przepływu danych .....	134
Podsumowanie .....	144

<b>II Dane rozproszone .....</b>	<b>151</b>
Skalowanie pod kątem wyższego obciążenia	151
<b>5. Replikacja .....</b>	<b>157</b>
Liderzy i obserwatorzy	158
Problemy z opóźnieniem replikacji	166
Replikacja z wieloma liderami	171
Replikacja bez lidera	180
Podsumowanie	193
<b>6. Podział na partycje .....</b>	<b>201</b>
Podział na partycje i replikacja	202
Podział na partycje danych typu klucz-wartość	203
Podział na partycje a indeksy pomocnicze	207
Równoważenie partycji	210
Trasowanie żądań	214
Podsumowanie	216
<b>7. Transakcje .....</b>	<b>223</b>
Niejasne pojęcie transakcji	224
Niskie poziomy izolacji	233
Sekwencyjność	250
Podsumowanie	263
<b>8. Problemy z systemami rozproszonymi .....</b>	<b>271</b>
Błędy i awarie częściowe	272
Zawodne sieci	274
Zawodne zegary	283
Wiedza, prawda i kłamstwa	295
Podsumowanie	304
<b>9. Spójność i konsensus .....</b>	<b>315</b>
Gwarancje spójności	316
Liniowość	317
Gwarancje uporządkowania	331
Transakcje rozproszone i konsensus	343
Podsumowanie	361

<b>III Dane pochodne .....</b>	<b>375</b>
Systemy zapisu a systemy danych pochodnych	375
Przegląd rozdziałów	376
<b>10. Przetwarzanie wsadowe .....</b>	<b>379</b>
Przetwarzanie wsadowe z użyciem narzędzi unixowych	380
MapReduce i rozproszone systemy plików	386
Poza model MapReduce	406
Podsumowanie	415
<b>11. Przetwarzanie strumieniowe .....</b>	<b>425</b>
Przesyłanie strumieni zdarzeń	426
Strumień a bazy danych	436
Przetwarzanie strumieniowe	447
Podsumowanie	462
<b>12. Przyszłość systemów danych .....</b>	<b>473</b>
Integrowanie danych	474
Podział baz danych na komponenty	482
Dążenie do poprawności	497
Robienie tego, co słuszne	513
Podsumowanie	522
<b>Słowniczek .....</b>	<b>533</b>
<b>Skorowidz .....</b>	<b>542</b>



---

# Przedmowa

Jeśli w ostatnich latach pracowałeś jako inżynier oprogramowania (zwłaszcza w obszarze systemów zaplecza i działających po stronie serwera), prawdopodobnie byłeś bombardowany wieloma modnymi określeniami związanymi z przechowywaniem i przetwarzaniem danych. NoSQL. Bit Data. Web-scale. Dzielenie danych (ang. *sharding*). Spójność ostateczna (ang. *eventual consistency*). ACID. Twierdzenie CAP. Usługi działające w chmurze. MapReduce. Praca w czasie rzeczywistym.

W ostatnim dziesięcioleciu pojawiło się wiele interesujących nowinek w dziedzinie baz danych, systemów rozproszonych i metod budowania aplikacji z ich użyciem. Na powstawanie tych nowinek wpływa wiele czynników:

- Firmy internetowe (takie jak Google, Yahoo!, Amazon, Facebook, LinkedIn, Microsoft i Twitter) obsługują duże ilości danych i ruchu internetowego, co zmusza je do tworzenia nowych narzędzi pozwalających wydajnie radzić sobie z działaniami na dużą skalę.
- Firmy muszą być elastyczne, niskim kosztem testować hipotezy i szybko reagować na nowe informacje z rynku dzięki krótkiemu cyklowi rozwoju oprogramowania i ułatwiającym zmiany modelom danych.
- Oprogramowanie bezpłatne i o otwartym dostępie do kodu źródłowego okazało się sukcesem i w wielu środowiskach jest obecnie wybierane zamiast rozwiązań komercyjnych lub rozwijanych wewnątrz firmy.
- Szybkość taktowania procesorów rośnie powoli, jednak standardem są procesory wielordzeniowe, a sieci stają się coraz szybsze. Oznacza to, że popularność przetwarzania równoległego będzie rosła.
- Mimo że pracujesz w niewielkim zespole, możesz budować systemy rozproszone działające na wielu komputerach, a nawet w wielu obszarach geograficznych. Jest to możliwe dzięki rozwiązaniom IaaS (ang. *infrastructure as a service*, czyli infrastruktura jako usługa) takim jak Amazon Web Services.
- Od wielu usług oczekuje się obecnie, że będą wysoce dostępne. Przedłużające się przestoje spowodowane awariami lub konserwacją są akceptowane w coraz mniejszym stopniu.

*Aplikacje intensywnie przetwarzające dane* wymagają przesuwania granic tego, co stało się możliwe, dzięki wykorzystaniu nowinek technologicznych. Aplikacja jest nazywana *intensywnie przetwarzającą dane*, jeśli podstawowy problem stanowią w niej dane (może tu chodzić o ich ilość, złożoność lub szybkość zmian). Inną kategorią są aplikacje *wymagające dużej mocy obliczeniowej*, gdzie wąskie gardło stanowi szybkość pracy procesora.

Narzędzia i transakcje, które wspomagają aplikacje intensywnie przetwarzające dane w przechowywaniu i przetwarzaniu informacji, są szybko dostosowywane do zachodzących zmian. Dużo uwagi zyskały systemy bazodanowe nowego rodzaju (NoSQL), jednak bardzo ważne są też kolejki komunikatów, pamięć podręczna, indeksy używane do wyszukiwania, platformy przetwarzania wsadowego i strumieniowego oraz powiązane technologie. W wielu aplikacjach używane są kombinacje takich rozwiązań.

Modne słówka pojawiające się w tym obszarze są oznaką entuzjazmu wobec nowych możliwości. To świetnie, jednak inżynierowie i architekci oprogramowania potrzebują poprawnego technicznie i precyzyjnego zrozumienia różnych technologii oraz związanych z nimi kompromisów, jeśli mają budować solidne aplikacje. To zrozumienie wymaga sięgnięcia głębiej niż do modnych słówek.

Na szczęście obok szybkich zmian w technologii istnieją też trwałe zasady, które pozostają prawdziwe niezależnie od używanej wersji danego narzędzia. Jeśli rozumiesz te zasady, możesz określić przeznaczenie każdego narzędzia, a także zobaczyć, jak poprawnie z niego korzystać i jak unikać związanych z nim pułapek. Ta książka Ci w tym pomoże.

Zadanie tej publikacji to ułatwienie poruszania się po zróżnicowanym i szybko zmieniającym się obszarze technologii przetwarzania i przechowywania danych. Ta książka nie jest samouczkiem dotyczącym konkretnego narzędzia. Nie jest to też podręcznik pełen suchej teorii. Zamiast tego znajdziesz tu przykłady skutecznych systemów danych — technologii, które stanowią podstawę wielu popularnych aplikacji i każdego dnia muszą spełniać w środowisku produkcyjnym wymogi z zakresu skalowalności, wydajności i niezawodności.

Zapoznasz się z wewnętrznymi mechanizmami tych systemów, analizami najważniejszych algorytmów, omówieniem zasad działania i koniecznymi kompromisami. Przy okazji staram się przedstawić przydatne sposoby *myślenia o* systemach danych. Zobaczysz nie tylko, *jak* te systemy działają, ale też *dlaczego* funkcjonują właśnie tak i jakie pytania na ich temat należy zadać.

Po przeczytaniu tej książki będziesz potrafił określić, jakiego rodzaju technologie są odpowiednie do poszczególnych zastosowań, a także zrozumieć, jak łączyć narzędzia, aby uzyskać podstawy solidnej architektury aplikacji. Nie będziesz gotów do zbudowania własnego systemu bazodanowego od podstaw, ale na szczęście rzadko jest to potrzebne. Rozwiniesz jednak dobre intuicyjne zrozumienie tego, jak systemy działają na zapleczu, dzięki czemu będziesz mógł analizować ich pracę, podejmować trafne decyzje projektowe i wyszukiwać źródła problemów, które się pojawiają.

## Kto powinien przeczytać tę książkę?

Jeśli rozwijasz aplikacje obejmujące serwer lub moduł zaplecza do przechowywania lub przetwarzania danych i te aplikacje korzystają z internetu (są to np. aplikacje sieciowe lub mobilne albo podłączone do internetu czujniki), jest to książka dla Ciebie.

To pozycja dla inżynierów oprogramowania, architektów oprogramowania i menedżerów technicznych, którzy uwielbiają programować. Jest ona odpowiednia zwłaszcza w sytuacji, gdy trzeba podejmować decyzje na temat architektury systemów — np. wybrać narzędzia do rozwiązywania danego problemu i określić, jak w optymalny sposób je zastosować. Jednak nawet wtedy, gdy nie masz możliwości wyboru narzędzi, ta książka pomoże Ci lepiej zrozumieć ich mocne i słabe strony.



Powinieneś mieć doświadczenie w budowaniu aplikacji internetowych lub usług sieciowych. Powinieneś też znać relacyjne bazy danych i SQL. Znajomość baz nierelacyjnych i innych narzędzi związanych z danymi to plus, nie jest to jednak konieczne. Pomocna będzie ogólna wiedza na temat standardowych protokołów sieciowych takich jak TCP i HTTP. W kontekście tej książki używane języki programowania i platformy nie mają znaczenia.

Jeśli którekolwiek z poniższych stwierdzeń jest prawdziwe, ta publikacja będzie dla Ciebie wartościowa:

- Chcesz się nauczyć budować skalowalne systemy przetwarzania danych — np. w celu obsługi aplikacji internetowych lub mobilnych mających miliony użytkowników.
- Musisz tworzyć aplikacje, które są wysoce dostępne (minimalizowanie przestojów) i opcjonalnie odporne na problemy.
- Szukasz sposobów na to, by systemy długoterminowo były łatwiejsze w konserwacji — nawet jeśli będą się rozrastać i gdy zmieniają się wymagania lub technologie.
- Masz wrodzoną ciekawość tego, jak rzeczy działają, i chcesz wiedzieć, jak pracują popularne witryny oraz usługi internetowe. W tej książce przeanalizowane są wewnętrzne mechanizmy różnych baz i systemów przetwarzania danych. Dużą przyjemność sprawia przyglądanie się błyskotliwym pomysłom związanym z projektami baz oraz systemów przetwarzania danych.

W kontekście omawiania skalowalnych systemów przetwarzania danych ludzie czasem mówią rzeczy w rodzaju: „Wasza firma to nie Google ani Amazon. Przestańcie się martwić skalowaniem i po prostu zastosujcie relacyjną bazę danych”. Jest w tym trochę prawdy. Budowanie rozwiązań na niepotrzebną skalę to marnowanie wysiłku i może prowadzić do projektu, które nie pozwala na zmiany. To forma przedwczesnej optymalizacji. Jednak ważne jest też to, by dobrać narzędzie odpowiednie do zadania, a różne technologie mają specyficzne wady i zalety. Zobaczysz, że relacyjne bazy danych są ważnym, ale nie ostatnim słowem w obszarze obsługi danych.

## Zakres tej książki

Ta pozycja nie ma zapewniać szczegółowych instrukcji na temat instalowania lub używania konkretnych pakietów oprogramowania lub interfejsów API. Istnieje już obfita dokumentacja takich narzędzi. Zamiast tego omówione są tu różne zasady i kompromisy związane z systemami przetwarzania danych. Opisane są też decyzje projektowe podejmowane w rozmaitych produktach.

Wszystkie odsyłacze z książki zostały sprawdzone na etapie prac nad publikacją, jednak natura internetu sprawia, że odnośniki niestety często przestają działać. Jeśli natrafisz na niedziałający odsyłacz, możesz próbować namierzyć dane materiały w wyszukiwarce. Prac naukowych możesz poszukać w wyszukiwarce Google Scholar, aby znaleźć otwarte pliki PDF. Inna możliwość to sprawdzenie literatury cytowanej na stronie <https://github.com/ept/ddia-references>, gdzie są dostępne aktualne odsyłacze.

Omawiana jest tu głównie *architektura* systemów przetwarzania danych i sposoby integrowania ich w aplikacje intensywnie przetwarzające dane. W tej książce nie ma miejsca na opis wdrażania, eksploatacji, zabezpieczeń, zarządzania i innych obszarów. Są to złożone i ważne zagadnienia, których nie da się rzetelnie przedstawić „na marginesie” tej publikacji. Zaslugują one na odrębne tomy.

Wiele technologii opisywanych w tej książce zalicza się do obszaru związanego z modnym określeniem *Big Data*. Jednak pojęcie „Big Data” jest tak często nadużywane i tak słabo zdefiniowane, że nie przydaje się w poważnych dyskusjach inżynieryjnych. W tej pozycji używane są mniej wieloznaczne nazwy, np. systemy z jednym węzłem i systemy rozproszone oraz systemy online (interaktywne) i systemy offline (przetwarzania wsadowego).

W tej książce faworyzuje się oprogramowanie bezpłatne i o otwartym dostępie do kodu źródłowego (ang. *free and open source software* — **FOSS**), ponieważ czytanie, modyfikowanie i wykonywanie kodu źródłowego to świetny sposób na dokładne zrozumienie, jak coś działa. Otwarte platformy zmniejszają też ryzyko uzależnienia się od producenta. Jednak tam, gdzie to wskazane, opisano też zastrzeżone oprogramowanie (bez dostępu do kodu źródłowego, rozwiązania w modelu SaaS lub wewnętrznie rozwijany kod tylko opisywany w literaturze, ale niedostępniany publicznie).

## Struktura książki

Tekst został podzielony na trzy części:

1. W części I opisane są najważniejsze pomysły będące podstawą projektu aplikacji intensywnie przetwarzających dane. Najpierw w rozdziale 1. omówione są cele, jakie programiści starają się osiągnąć: niezawodność, skalowalność i łatwość konserwacji. Dowiesz się, jak należy o nich myśleć i jak je osiągać. Rozdział 2. zawiera porównanie kilku różnych modeli danych i języków obsługi zapytań. Zobaczysz, jak je dobierać do różnych sytuacji. Rozdział 3. to omówienie systemów przechowywania. Dowiesz się, jak bazy porządkują dane na dysku, aby można je potem wydajnie wyszukiwać. Rozdział 4. dotyczy formatów kodowania danych (serializacji) i zmian schematów w czasie.
2. Część II to przejście od danych przechowywanych na jednym komputerze do danych rozproszonych między wiele maszyn. Taka zmiana jest często niezbędna, aby uzyskać skalowalność, jednak wiąże się z różnymi wyjątkowymi wyzwaniami. Najpierw opisane są: replikacja (rozdział 5.), dzielenie danych (rozdział 6.) i transakcje (rozdział 7.). Dalej znajdziesz szczegółowe omówienie problemów z systemami rozproszonymi (rozdział 8.) i dowiesz się, na czym polega osiągnięcie spójności i konsensusu w systemach rozproszonych (rozdział 9.).
3. Część III zawiera omówienie systemów uzyskujących zbiory danych pochodnych na podstawie innych zbiorów danych. Dane pochodne często się pojawiają w systemach niejednorodnych. Gdy nie istnieje jedna baza, która potrafi wszystko robić prawidłowo, aplikacje muszą integrować kilka różnych baz, pamięci podręcznych, indeksów itd. W rozdziale 10. opisano przetwarzanie wsadowe danych pochodnych. Rozwinięcie tego podejścia stanowi przetwarzanie strumieniowe omówione w rozdziale 11. Na zakończenie w rozdziale 12. wszystkie elementy są łączone ze sobą. Opisane są tam metody przyszłości służące do budowania niezawodnych, skalowalnych i łatwych w konserwacji aplikacji.

# Literatura cytowana i dodatkowa

Większość wiedzy z tej książki została już przedstawiona w różnych postaciach: w prezentacjach na konferencjach, pracach naukowych, artykułach z blogów, kodzie, systemach śledzenia błędów, listach dyskusyjnych i tradycji inżynierskiej. Ta książka stanowi podsumowanie najważniejszych pomysłów z wielu różnych źródeł. W tekście znajdziesz odniesienia do źródeł informacji. Literatura cytowana podawana w końcowej części każdego rozdziału stanowi doskonałe źródło wiedzy, jeśli chcesz szczegółowo zapoznać się z danym obszarem. Większość podawanych tekstów jest dostępna bezpłatnie w internecie.

## Podziękowania

Ta książka to połączenie i usystematyzowanie wiedzy oraz licznych pomysłów innych osób. Uwzględniłem tu doświadczenia zarówno z badań akademickich, jak i z praktyki w branży przemysłowej. W informatyce często pociągają nas nowe i modne rozwiązania, uważam jednak, że możemy bardzo dużo się nauczyć na podstawie tego, co zostało zrobione wcześniej. Ta książka zawiera ponad 800 odwołań do artykułów, tekstów z blogów, wystąpień, dokumentacji itd. Były one dla mnie nieocenionym źródłem informacji w trakcie nauki. Jestem bardzo wdzięczny autorom tych materiałów za to, że podzielili się swoją wiedzą.

Dużo nauczyłem się także z rozmów z wieloma osobami, które poświęciły swój czas na dyskusje nad pomysłami lub cierpliwe wyjaśnianie mi różnych rzeczy. Oto osoby, do których czuję największą wdzięczność: Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Helland, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O’Neil, Onora O’Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrer, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden i Brett Wooldridge.

Inne osoby były bardzo pomocne w trakcie pisania tej książki, recenzując wersje wstępne i przekazując mi informacje zwrotne. Oto ludzie, którym jestem za to szczególnie wdzięczny: Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes i Ben Summers. Oczywiście to ja ponoszę odpowiedzialność za wszystkie niewykryte błędy i niestrawne opinie z tej książki.

Za pomoc w tym, by ta książka ujrzała światło dzienne, i za cierpliwość w stosunku do mojego wolnego tempa pisania i nietypowych żądań dziękuję moim redaktorom, Marie Beaugureau, Mike’owi Loukidesowi i Ann Spencer, oraz całemu zespołowi z wydawnictwa O’Reilly. Za pomoc w znaj-

dowaniu właściwych słów składam podziękowania Rachel Head. Za czas i możliwość pisania mimo innych obowiązków w pracy jestem wdzięczny Alastairowi Beresfordowi, Susan Goodhue, Nehi Narkhede i Kevinowi Scottowi.

Specjalne podziękowania należą się Shabbirowi Diwanowi i Ediemu Freedmanowi, którzy z wielką starannością opracowali mapy dołączone do rozdziałów. To wspaniałe, że podchwycili niekonwencjonalny pomysł rysowania map i sprawili, że stały się one tak piękne i atrakcyjne.

Na zakończenie kieruję wyrazy miłości do przyjaciół i rodziny, bez których nie potrafiłbym przejść przez trwający prawie cztery lata proces pisania. Jesteście najlepsi!

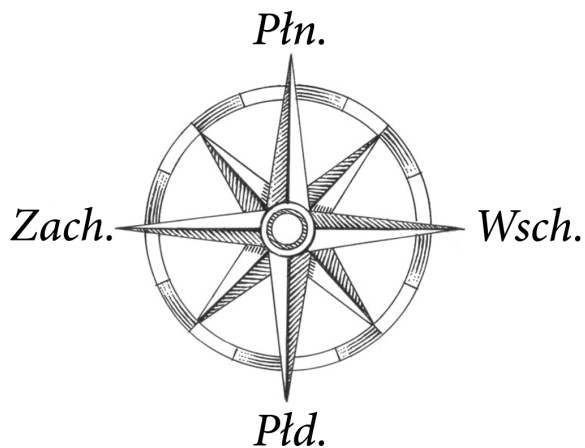
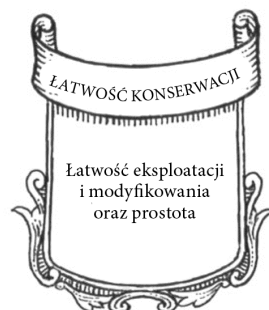
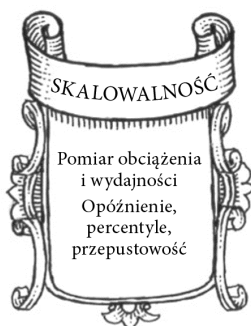
---

# Podstawy systemów danych

Pierwsze cztery rozdziały dotyczą najważniejszych pomysłów związanych z wszystkimi systemami danych — i tych działających na jednej maszynie, i tych rozproszonych w klastrze:

1. W rozdziale 1. przedstawiono terminologię i podejście używane w tej książce. Wyjaśniono tu, co oznaczają takie słowa jak *niezawodność*, *skalowalność* i *łatwość konserwacji*. Dowiesz się też, jak można próbować osiągnąć te cele.
2. Rozdział 2. zawiera porównanie kilku różnych modeli danych i języków obsługi zapytań. Te elementy to najbardziej widoczne czynniki różnicujące bazy danych z perspektywy programistów. Zobaczysz tu, że w poszczególnych sytuacjach odpowiednie są różne modele.
3. W rozdziale 3. omówiono wewnętrzne mechanizmy systemów składowania danych. Opisano tu też, w jaki sposób bazy rozmieszczają dane na dysku. Różne systemy składowania danych są zoptymalizowane pod kątem rozmaitych rodzajów obciążenia roboczego. Wybór właściwego systemu może mieć bardzo duży wpływ na wydajność.
4. Rozdział 4. zawiera porównanie rozmaitych formatów kodowania danych (serializacji) z naciskiem na to, jak radzą sobie one w środowisku, w którym wymagania stawiane aplikacji się zmieniają i konieczne jest dostosowywanie do nich schematów.

Dalej, w części II, omówione są konkretne zagadnienia związane z rozproszonymi systemami danych.



# Niezawodne, skalowalne i łatwe w konserwacji aplikacje

*Internet został opracowany tak dobrze, że większość ludzi traktuje go jak zasoby naturalne takie jak Pacyfik, a nie jak coś zbudowanego przez człowieka.*

*Kiedy po raz ostatni stworzono wolną od usterek technologię na tak dużą skalę?*

— Alan Kay, z wywiadu dla magazynu „Dr Dobbs’s Journal” (2012)

Obecnie wiele aplikacji to rozwiązania *intensywnie przetwarzające dane* (ang. *data-intensive*), a nie *wymagające obliczeniowo* (ang. *compute-intensive*). W tego rodzaju aplikacjach sama moc procesora rzadko jest czynnikiem ograniczającym. Większymi problemami są zwykle: ilość danych, ich złożoność i szybkość, z jaką się zmieniają.

Aplikacje intensywnie przetwarzające dane są zwykle zbudowane ze standardowych cegiełek, które zapewniają często potrzebne mechanizmy. Na przykład wiele aplikacji musi:

- przechowywać dane, aby później dana aplikacja (lub inna) mogła je znaleźć (*baza danych*);
- zapamiętywać wyniki kosztownych operacji, aby przyspieszyć odczyt (*pamięć podręczna*);
- umożliwiać użytkownikom wyszukiwanie danych na podstawie słów kluczowych lub filtrów (*indeksy wyszukiwania*);
- przysyłać do innego procesu komunikat w celu jego asynchronicznej obsługi (*przetwarzanie strumieniowe*);
- okresowo przetwarzać duże ilości zakumulowanych danych (*przetwarzanie wsadowe*).

Jeśli brzmi to aż nazbyt oczywiście, to dlatego, że tego typu *systemy danych* są tak udaną abstrakcją. Posługujemy się nimi przez cały czas, nie myśląc o tym wiele. W trakcie tworzenia aplikacji większość inżynierów nie myśli nawet o pisaniu od podstaw nowego systemu składowania danych, ponieważ bazy doskonale nadają się do obsługi danych.

Jednak rzeczywistość jest bardziej złożona. Istnieje wiele systemów bazodanowych o różnych cechach, ponieważ poszczególne aplikacje mają odmienne wymagania. Istnieje też wiele metod obsługi pamięci podręcznej, kilka sposobów tworzenia indeksów wyszukiwania itd. W trakcie tworzenia aplikacji trzeba określić, które narzędzia i podejścia będą najbardziej odpowiednie do wykonywania danego zadania. Ponadto czasem trudno jest połączyć komponenty, gdy trzeba zrobić coś, z czym pojedyncze narzędzie sobie nie radzi.

Ta książka to podróż po zasadach i praktycznych aspektach systemów danych oraz po wykorzystywaniu ich do budowania aplikacji intensywnie przetwarzających dane. Zobaczysz tu podobieństwa i różnice między poszczególnymi narzędziami. Dowiesz się też, z czego wynikają ich cechy.

Ten rozdział zaczyna się od przeglądu podstawowych cech, jakie programiści starają się uzyskać: niezawodności, skalowalności i łatwości konserwacji systemów danych. Wyjaśniono tu, co oznaczają te cechy. Przedstawione są też sposoby myślenia o nich i podstawy potrzebne w dalszych rozdziałach. W następnych rozdziałach omawiane są kolejne warstwy. Poznasz tam różne decyzje projektowe, które trzeba uwzględnić w trakcie pracy nad aplikacjami intensywnie przetwarzającymi dane.

## Myślenie o systemach danych

Bazy danych, kolejki, pamięć podręczna itd. są zwykle traktowane jako narzędzia należące do zupełnie różnych kategorii. Choć baza danych i kolejka komunikatów pozornie są podobne (oba te narzędzia przechowują dane przez pewien czas), cechują się zdecydowanie odmiennymi wzorcami dostępu. To oznacza inną charakterystykę pracy, a tym samym i bardzo odmienne implementacje.

Dlaczego więc wszystkie takie rozwiązania są łączone pod zbiorczym określeniem *systemy danych*?

W ostatnich latach pojawiło się wiele nowych narzędzi do przechowywania i przetwarzania danych. Są one zoptymalizowane pod kątem różnych przypadków użycia i nie wpasowują się w tradycyjne kategorie [1]. Istnieją np. magazyny danych używane też jako kolejki komunikatów (Redis), a także kolejki komunikatów z gwarancjami trwałości typowymi dla baz danych (Apache Kafka). Granice między kategoriami się zacierają.

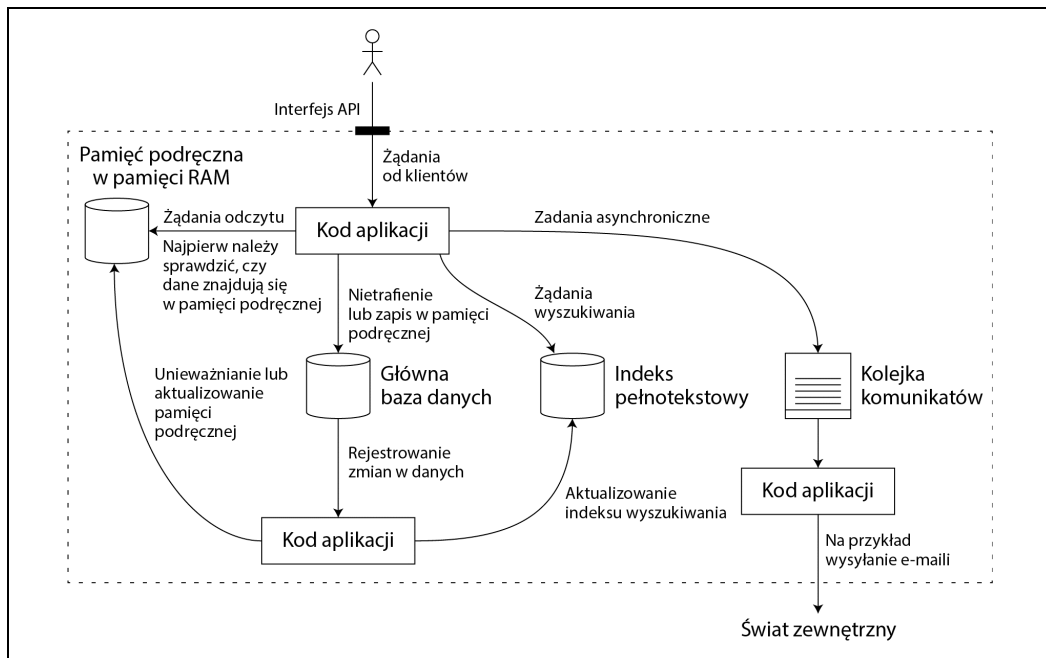
Ponadto coraz większa liczba aplikacji cechuje się tak wysokimi lub szerokimi wymaganiami, że żadne pojedyncze narzędzie nie potrafi zaspokoić wszystkich potrzeb z zakresu przetwarzania i składowania danych. Dlatego pracę dzieli się na zadania, które *mogą* być wydajnie wykonywane przez pojedyncze narzędzie, a różne narzędzia są łączone za pomocą kodu aplikacji.

Na przykład jeśli korzystasz z zarządzanej przez aplikację warstwy pamięci podręcznej (wykorzystującej system Memcached lub podobny) lub serwera wyszukiwania pełnotekstowego (takiego jak Elasticsearch lub Solr) oddzielnie od głównej bazy, zwykle to kod aplikacji odpowiada za synchronizowanie pamięci podręcznej i indeksów z podstawową bazą. Na rysunku 1.1 pokazano prosty przykład tego, jak może to wyglądać (szczegóły znajdziesz w dalszych rozdziałach).

Gdy łączysz kilka narzędzi, aby udostępnić usługę, interfejs usługi lub interfejs **API** (ang. *application programming interface*) zwykle ukrywają szczegóły implementacji przed klientem. Oznacza to utworzenie nowego systemu danych o specjalnym przeznaczeniu na podstawie mniejszych komponentów o przeznaczeniu ogólnym. Złożony system danych może zapewniać określone gwarancje dotyczące np. poprawnego unieważniania lub aktualizowania pamięci podręcznej po zapisie danych, tak aby klienci zewnętrzne otrzymywały spójne wyniki. W ten sposób stajesz się nie tylko programistą aplikacji, ale też projektantem systemu danych.

Jeśli projektujesz system danych lub usługę, musisz odpowiedzieć na wiele skomplikowanych pytań. Jak zapewnisz, że dane pozostaną prawidłowe i kompletne, nawet gdy wystąpią wewnętrzne błędy? Jak zapewnisz klientom stabilnie wysoką wydajność, nawet gdy wystąpią problemy w komponentach systemu? Jak system będzie się skalował, aby obsłużyć wzrost obciążenia? Jak powinien wyglądać właściwy interfejs API usługi?





Rysunek 1.1. Możliwa architektura systemu danych łączącego kilka komponentów

Na projekt systemu danych wpływa wiele czynników, w tym umiejętności i doświadczenie jego twórców, zależności od starszych systemów, termin zakończenia prac, odporność organizacji na różnego rodzaju ryzyka, ograniczenia prawne itd. Te czynniki są wysoce zależne od sytuacji.

W tej książce uwzględniane są głównie trzy kwestie ważne w większości systemów informatycznych:

#### Niezawodność

System powinien działać *prawidłowo* (poprawnie wykonywać swoje funkcje z oczekiwaną wydajnością) nawet w obliczu *problemów* (awarii sprzętowych lub programowych, a nawet błędów ludzkich). Zob. punkt „Niezawodność”.

#### Skalowalność

Gdy system się *rozrasta* (ze względu na ilość danych lub ruchu albo złożoność), powinny istnieć sensowne sposoby radzenia sobie z tym wzrostem. Zob. punkt „Skalowalność”.

#### Łatwość konserwacji

W przyszłości nad systemem pracować będzie wiele osób (inżynierów i pracowników operacyjnych zajmujących się zarówno konserwacją aktualnych funkcji, jak i dostosowywaniem systemu do nowych przypadków użycia). Wszyscy oni powinni móc pracować nad nim w *produktywny* sposób. Zob. punkt „Łatwość konserwacji”.

Te słowa są często używane bez dokładnego zrozumienia ich znaczenia. Reszta rozdziału została przeznaczona na rozważania o niezawodności, skalowalności i łatwości konserwacji. Później, w dalszych rozdziałach, przedstawione są różne techniki, architektury i algorytmy używane do osiągnięcia tych celów.

# Niezawodność

Każdy intuicyjnie rozumie, co oznacza, że coś jest niezawodne lub zawodne. Oto typowe oczekiwania z obszaru oprogramowania:

- Aplikacja wykonuje zadania żądane przez użytkownika.
- Aplikacja jest odporna na pomyłki popełnione przez użytkownika lub korzystanie z niej w niezaplanowany sposób.
- Wydajność aplikacji jest wystarczająca dla oczekiwanego sposobu użytkowania przy spodziewanym obciążeniu i ilości danych.
- System zapobiega nieuprawnionemu dostępowi i nadużyciom.

Jeśli wszystkie te rzeczy razem oznaczają „poprawne działanie”, *niezawodność* można w przybliżeniu rozumieć jako „ciągłe poprawne działanie nawet po wystąpieniu problemów”.

Rzeczy, które mogą pójść źle, są nazywane *błędami*, a systemy przewidujące błędy i radzące sobie z nimi określa się mianem *odpornych na błędy*. To ostatnie pojęcie może być nieco mylące, ponieważ sugeruje, że można zbudować system odporny na błędy wszelkiego rodzaju, co w praktyce jest niewykonalne. Gdyby cała ziemia (wraz z wszystkimi serwerami) została wchłonięta przez czarną dziurę, zapewnienie odporności na taki „błąd” wymagałoby hostingu rozwiązania w kosmosie. Życzę powodzenia w pozyskiwaniu w budżecie środków na taki system! Dlatego sensowne jest mówienie tylko o odporności na błędy *określonych rodzajów*.

Zauważ, że błąd (ang. *fault*) to nie to samo co awaria (ang. *failure*) [2]. Błąd zwykle definiuje się jako odstępstwo jednego komponentu systemu od specyfikacji, natomiast *awaria* następuje, gdy system jako całość przestaje świadczyć oczekiwane usługi użytkownikowi. Nie da się ograniczyć ryzyka wystąpienia błędów do zera. Dlatego zwykle najlepiej jest projektować mechanizmy zapewniania odporności na błędy, aby nie dopuścić do tego, by błąd doprowadził do awarii. W tej książce opisano kilka metod budowania niezawodnych systemów z użyciem zawodnych elementów.

Sprzeczne z intuicją jest to, że w tego typu odpornych na błędy systemach sensowne może być *zwiększanie* liczby błędów przez celowe ich wywoływanie (np. przez losowe zamykanie poszczególnych procesów bez ostrzeżenia). Wiele błędów krytycznych wynika ze złej obsługi błędów [3]. Celowo wprowadzając błędy, gwarantujesz, że mechanizmy zapewniania odporności są stale sprawdzane i testowane. Zwiększa to pewność, że błędy zostaną poprawnie obsłużone, gdy wystąpią z przyczyn naturalnych. Przykładem zastosowania tego podejścia jest mechanizm *Chaos Monkey* [4] opracowany przez firmę Netflix.

Choć zwykle preferuje się zapewnianie odporności na błędy, a nie zapobieganie im, w niektórych sytuacjach zapobieganie okazuje się lepsze niż leczenie (np. wtedy, gdy nie istnieje lekarstwo). Dotyczy to np. kwestii bezpieczeństwa. Jeśli napastnik włamie się do systemu i uzyska dostęp do poufnych danych, nie da się odwrócić skutków tego zdarzenia. Jednak w tej książce opisywane są głównie błędy, których skutkom można zaradzić. Błędy te są opisane w następnych punktach.

## Błędy sprzętowe

Gdy zastanawiasz się nad awariami systemów, na myśl przychodzą błędy sprzętowe. Awarie dysków twardych, błędy w pamięci RAM, przerwy w dostawie prądu, odłączenie niewłaściwego kabla sieciowego. Każdy, kto pracował w dużym centrum danych, potwierdzi, że gdy liczba maszyn jest duża, tego typu sytuacje zdarzają się *nieustannie*.

Średni czas do awarii (ang. *mean time to failure* — **MTTF**) dysku twardego wynosi od ok. 10 do 50 lat [5, 6]. Dlatego w klastrze składowania danych obejmującym 10 tys. dysków należy oczekiwać, że dziennie średnio zepsuje się jeden dysk.

Pierwszą reakcją jest zwykle zapewnienie nadmiarowości poszczególnych komponentów sprzętowych, aby zmniejszyć częstotliwość awarii systemu. Można łączyć dyski w macierze RAID, stosować dwa źródła zasilania i wymienne procesory dla serwerów, a także używać baterii i spalinyowych generatorów prądu, aby zapewnić zasilanie awaryjne centrum danych. Gdy jeden komponent przestanie działać, nadmiarowy zajmie jego miejsce na czas wymiany uszkodzonej jednostki. To podejście nie pozwala w pełni zapobiec awariom z powodu problemów sprzętowych, jednak jest dobrze znane i często sprawia, że maszyna potrafi latami pracować bez przestojów.

Do niedawna w większości aplikacji nadmiarowość komponentów sprzętowych wystarczała, ponieważ sprawiała, że całkowita awaria maszyny zdarzała się stosunkowo rzadko. Jeśli potrafisz szybko odtworzyć kopię zapasową na nowej maszynie, przestój wynikający z awarii w większości systemów nie jest katastrofą. Dlatego nadmiarowość w postaci dodatkowych maszyn może być potrzebna tylko w nielicznych zastosowaniach, w których wysoka dostępność była absolutnie konieczna.

Jednak wraz ze wzrostem ilości danych i wymaganiami obliczeniowymi aplikacji coraz częściej aplikacje potrzebowały większej liczby maszyn, co skutkowało proporcjonalnym wzrostem liczby błędów sprzętowych. Ponadto w niektórych platformach do udostępniania chmury (np. w usłudze Amazon Web Services — AWS) dość często zdarza się, że instancje maszyn wirtualnych stają się niedostępne bez ostrzeżenia [7], ponieważ platformy te są tak projektowane, by stawiały elastyczność i łatwość dostosowywania<sup>1</sup> nad niezawodność pojedynczych maszyn.

Dlatego następuje przechodzenie w kierunku systemów odpornych na utratę całych maszyn. W tym celu programowe metody zapewniania odporności na błędy są stosowane zamiast lub obok nadmiarowości sprzętu. Takie systemy mają też zalety operacyjne. System z jednym serwerem wymaga zaplanowanego przestoju, gdy potrzebujesz ponownie uruchomić maszynę (np. w celu wprowadzenia poprawek bezpieczeństwa systemu operacyjnego), natomiast system, który radzi sobie z awariami maszyn, można aktualizować węzeł po węźle bez powodowania przestoju całego systemu (jest to *aktualizacja stopniowa*; zob. rozdział 4.).

## Błędy programowe

Błędy sprzętowe zwykle są traktowane jako losowe i niezależne od siebie. Awaria dysku w jednej maszynie nie oznacza, że wystąpią problemy także z dyskiem innej maszyny. Mogą pojawić się pewne słabe korelacje (np. z powodu wspólnej przyczyny takiej jak temperatura w szafie serwerowej),

---

<sup>1</sup> Definicję znajdziesz w punkcie „Metody radzenia sobie z obciążeniem”.

jednak zwykle mało prawdopodobne jest, że duża liczba komponentów sprzętowych zawiedzie w tym samym momencie.

Inną kategorią są systematyczne błędy w systemie [8]. Trudniej je przewidzieć, a ponieważ występowanie takich problemów w węzłach jest skorelowane, błędy tego typu powodują znacznie więcej awarii systemu niż nieskorelowane błędy sprzętowe [5]. Oto przykłady takich problemów:

- Usterka oprogramowania powodująca, że każda instancja serwera aplikacji ulega awarii po otrzymaniu określonych niewłaściwych danych wejściowych. Pomyśl np. o sekundzie przestępczej 30 czerwca 2012 r., która sprawiła, że wiele aplikacji jednocześnie przestało działać z powodu usterki w jądrze Linuksa [9].
- Niekontrolowany proces zużywający wspólne zasoby (czas procesora, pamięć, przestrzeń dyskową lub przepustowość łącza).
- Usługa, od której zależy system, działa powoli, przestaje reagować lub zaczyna zwracać nieprawidłowe odpowiedzi.
- Awaria kaskadowa, w trakcie której niewielka usterka jednego komponentu skutkuje błędem w innym komponencie, co z kolei prowadzi do dalszych problemów [10].

Usterki, które powodują tego rodzaju błędy programowe, często pozostają w ukryciu przez długi czas — do momentu ujawnienia ich z powodu nietypowego zbiegu okoliczności. Wtedy okazuje się, że w oprogramowaniu przyjęto określone założenie dotyczące środowiska i choć zwykle to założenie jest prawdziwe, z jakiegoś powodu ostatecznie przestało takie być [11].

Nie istnieje proste rozwiązanie problemu błędów systematycznych w oprogramowaniu. Pomocnych może być wiele drobiazgów: staranne przemyślenie założeń dotyczących systemu i występujących interakcji, odizolowanie procesu, umożliwienie procesowi na awarię i restart, pomiar, monitorowanie i analizowanie działania systemu w środowisku produkcyjnym. Jeśli system ma zapewniać określone gwarancje (np. w kolejce komunikatów może to polegać na tym, że liczba komunikatów przychodzących musi być równa liczbie komunikatów wychodzących), może stale sprawdzać swój stan w trakcie pracy i zgłaszać alarm po wykryciu rozbieżności [12].

## Błędy ludzkie

To ludzie projektują i tworzą systemy oprogramowania. Ludźmi są też operatorzy sterujący pracą systemów. Nawet mając najlepsze intencje, ludzie popełniają pomyłki. Na przykład w badaniach nad dużą usługą internetową odkryto, że główną przyczyną przestojów były błędy w konfiguracji spowodowane przez operatorów. Błędy sprzętowe (serwerów lub sieci) odpowiadały tylko za 10 – 25% przestojów [13].

Jak sprawić, by systemy były niezawodne mimo ludzkich pomyłek? W najlepszych systemach stosuje się kilka podejść:

- Systemy są projektowane w taki sposób, aby zminimalizować możliwości spowodowania błędów. Na przykład odpowiednio zaprojektowane abstrakcje, interfejsy API i interfejsy administratora powodują, że łatwo jest robić „właściwe rzeczy”, a zniechęcają do robienia „niewłaściwych”. Jeśli jednak interfejs okaże się zbyt ograniczający, użytkownicy nie będą z niego korzystać, co oznacza utratę korzyści. Trudno więc zachować odpowiednią równowagę w tym obszarze.

- Izolowanie miejsc, w których ludzie popełniają najwięcej pomyłek, od miejsc, gdzie błędy mogą skutkować awariami. Przede wszystkim warto udostępnić kompletne nieprodukcyjne *środowisko izolowane* (ang. *sandbox*), w którym użytkownicy mogą bezpiecznie eksplorować możliwości i eksperymentować na rzeczywistych danych, ale bez wpływu na rzeczywistych użytkowników.
- Prowadzenie dokładnych testów na wszystkich poziomach: od testów jednostkowych po testy integracyjne i ręczne całego systemu [3]. Testy zautomatyzowane są powszechnie stosowane, dobrze zrozumiałe i wartościowe zwłaszcza w kontekście sprawdzania przypadków brzegowych, które rzadko występują w trakcie normalnej eksploatacji systemu.
- Umożliwianie szybkiego i łatwego przywracania stanu po błędach ludzkich w celu zminimalizowania ich wpływu, gdy wydarzy się awaria. Można np. umożliwić szybkie wycofywanie zmian w konfiguracji, stopniowo wprowadzać nowy kod (tak aby nieoczekiwane usterki dotyczyły tylko małego podzbioru użytkowników) i zapewniać narzędzia do wykonywania ponownych obliczeń na danych (jeśli się okaże, że wcześniejsze obliczenia były błędne).
- Przygotowanie szczegółowego i precyzyjnego systemu monitorowania, np. wskaźników wydajności i współczynników błędów. W innych dziedzinach inżynierii ten obszar nazywa się *telemetrią*. Gdy rakieta już wystartuje, telemetria jest niezbędna do śledzenia tego, co się dzieje, i do zrozumienia awarii [14]. Monitorowanie pozwala wcześniej wykryć sygnały ostrzegawcze i sprawdzać, czy jakieś założenia lub ograniczenia nie zostały naruszone. Gdy wystąpi problem, wskaźniki mogą być nieocenione w diagnozowaniu usterek.
- Wprowadzenie odpowiednich szkoleń i metod zarządzania. To skomplikowany i ważny temat wykraczający poza zakres tej książki.

## Jak ważna jest niezawodność?

Niezawodność jest istotna nie tylko w elektrowniach atomowych i oprogramowaniu do kontroli ruchu lotniczego. Także od bardziej zwyczajnych aplikacji oczekuje się, że będą działały niezawodnie. Usterki w aplikacjach biznesowych powodują utratę wydajności (i problemy prawne, jeśli dane są błędnie prezentowane), a przestoje sklepów elektronicznych mogą prowadzić do poważnych kosztów z powodu utraty dochodów i spadku reputacji.

Nawet w aplikacjach „niekrytycznych” programiści odpowiadają przed użytkownikami. Pomyśl o rodzicu, który przechowuje wszystkie zdjęcia i filmy swoich dzieci w służącym do tego programie [15]. Jak ta osoba się poczuje, jeśli baza danych zostanie nagle uszkodzona? Czy rodzic będzie wiedział, jak ją odzyskać na podstawie kopii zapasowej?

W niektórych sytuacjach zdarza się, że rezygnuje się z niezawodności na rzecz obniżenia kosztów rozwoju (np. w trakcie tworzenia prototypowego produktu na nieznanym rynku) lub kosztów operacyjnych (np. w usłudze o bardzo niskiej marży). Należy jednak zachować dużą ostrożność, wybierając drogę na skróty.

# Skalowalność

Nawet jeśli dziś system działa niezawodnie, nie oznacza to, że będzie tak w przyszłości. Jednym z typowych powodów spadku wydajności jest wzrost obciążenia. Możliwe, że liczba jednoczesnych użytkowników systemu wzrosła z 10 tys. do 100 tys. lub z miliona do 10 mln. Możliwe, że system przetwarza znacznie większą ilość danych niż wcześniej.

*Skalowalność* to pojęcie służące do opisu tego, czy system radzi sobie ze wzrostem obciążenia. Warto jednak zauważyć, że nie jest to jednowymiarowa etykieta, którą można nadać systemowi. Stwierdzenia typu „X się skaluje” lub „Y się nie skaluje” nic nie znaczą. Analizowanie skalowalności wymaga rozważenia pytań takich jak: „Jeśli system rozrośnie się w określony sposób, jakie będą możliwości poradzenia sobie z tym wzrostem?” i „Jak można dodać zasoby obliczeniowe, aby obsłużyć większe obciążenie?”.

## Opisywanie obciążenia

Najpierw należy zwięźle opisać aktualne obciążenie systemu. Dopiero potem można analizować kwestie związane ze wzrostem (co się stanie, jeśli obciążenie wzrośnie dwukrotnie?). Obciążenie można opisać za pomocą kilku liczb nazywanych *parametrami obciążenia*. Odpowiedni dobór tych parametrów zależy od architektury systemu. Może to być liczba żądań do serwera WWW na sekundę, stosunek odczytów do zapisów w bazie, liczba jednocześnie aktywnych użytkowników na czacie, współczynnik trafień pamięci podręcznej lub coś innego. Możliwe, że interesują Cię typowe przypadki. Możliwe też, że przyczyną występowania „wąskiego gardła” jest niewielka liczba skrajnych przypadków.

Aby przyjrzeć się temu w bardziej konkretny sposób, warto rozważyć działanie Twittera na podstawie danych opublikowanych w listopadzie 2012 r. [16]. Dwie podstawowe operacje na Twitterze to:

### *Publikowanie tweetów*

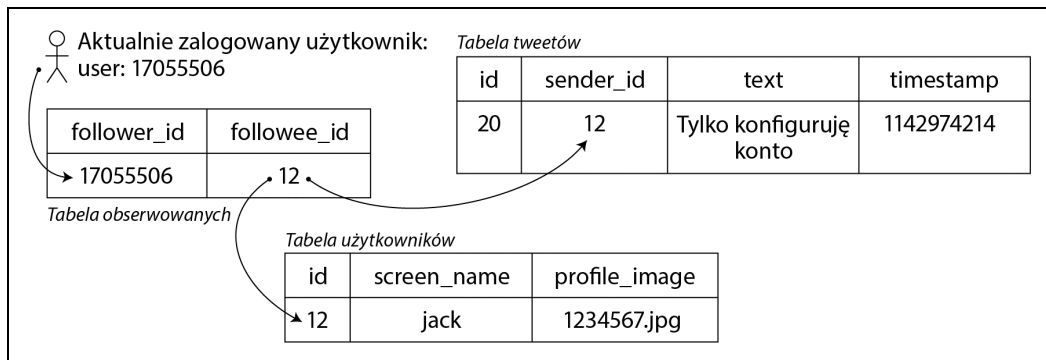
Użytkownik może opublikować nową wiadomość dla swoich obserwatorów (średnio 4,6 tys. żądań na sekundę; szczytowo ponad 12 tys. żądań na sekundę).

### *Wyświetlanie osi czasu*

Użytkownik może wyświetlić wiadomości opublikowane przez obserwowane osoby (300 tys. żądań na sekundę).

Sama obsługa 12 tys. zapisów na sekundę (szczytowa szybkość zamieszczania tweetów) jest stosunkowo prosta. Jednak w przypadku Twittera trudności ze skalowaniem nie wynikają przede wszystkim z liczby tweetów, ale z powodu *rozgałęzień* — każdy użytkownik obserwuje wielu innych i jest obserwowany przez liczne osoby. Są dwie ogólne metody wykonywania dwóch opisanych operacji:

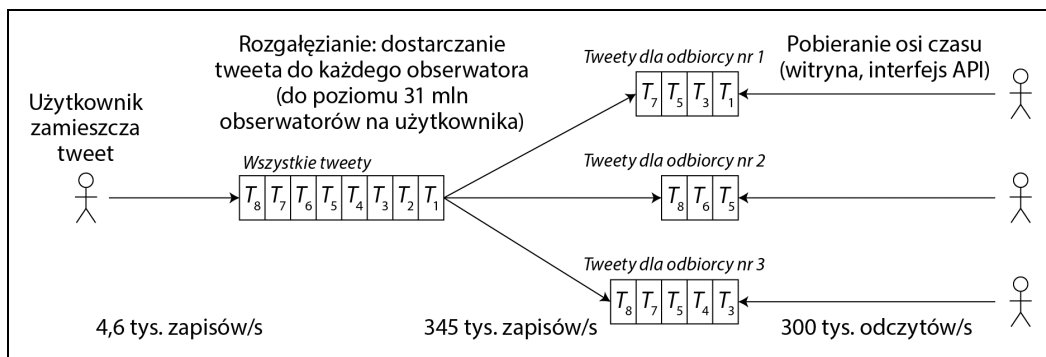
1. Zamieszczenie tweetu powoduje wstawienie go do globalnej kolekcji wiadomości. Gdy użytkownik żąda osi czasu, należy znaleźć wszystkie obserwowane przez niego osoby, odszukać wszystkie tweety każdej z nich i połączyć je (posortowane według czasu). W relacyjnej bazie danej, takiej jak na rysunku 1.2, można napisać zapytanie o następującej postaci:



Rysunek 1.2. Prosty schemat relacyjny służący do obsługi osi czasu na Twitterze

```
SELECT tweets.*, users.* FROM tweets
JOIN users ON tweets.sender_id = users.id
JOIN follows ON follows.followee_id = users.id
WHERE follows.follower_id = current_user
```

- Przechowywanie pamięci podręcznej z osią czasu każdego użytkownika (to coś w rodzaju skrzynki odbiorczej z tweetami każdego odbiorcy; zob. rysunek 1.3). Gdy użytkownik zamieszcza tweet, należy znaleźć wszystkich obserwatorów tej osoby i dodać ten nowy tweet do pamięci podręcznej z osią czasu każdego obserwatora. Żądanie odczytu osi czasu nie jest wtedy kosztowne, ponieważ wynik został wcześniej obliczony.



Rysunek 1.3. Potok danych służący do dostarczania tweetów do obserwatorów na Twitterze z poziomami obciążenia z listopada 2012 r. [16]

W pierwszej wersji Twittera używano podejścia nr 1, jednak systemom trudno było wtedy poradzić sobie z obciążeniem powodowanym przez zapytania pobierające oś czasu, dlatego firma wprowadziła podejście nr 2. Nowe podejście sprawdza się lepiej, ponieważ średnia szybkość publikowania tweetów jest o prawie dwa rzędy wielkości mniejsza niż częstotliwość odczytów osi czasu. Dlatego w tej sytuacji lepiej wykonywać więcej pracy na etapie zapisu i mniej w trakcie odczytu.

Wadę podejścia nr 2 stanowi jednak to, że zamieszczenie tweetu wymaga dużo dodatkowej pracy. Tweet jest średnio dostarczany do ok. 75 obserwatorów, dlatego 4,6 tys. tweetów na sekundę oznacza 345 tys. zapisów na sekundę w osiach czasu w pamięci podręcznej. Ta średnia ukrywa jednak to, że liczba obserwatorów na użytkownika jest bardzo zróżnicowana. Niektóre osoby mają ponad 30 mln

obserwatorów. To oznacza, że jeden tweet może prowadzić do ponad 30 mln zapisów w osiach czasu! Szybkie wykonywanie tej operacji (Twitter stara się dostarczać tweety obserwatorom w czasie 5 s) to poważne wyzwanie.

W przypadku Twittera rozkład liczby obserwatorów na użytkownika (możliwe, że z wagami zależnymi od tego, jak często użytkownicy publikują tweety) to ważny parametr obciążenia w trakcie analizy skalowalności, ponieważ określa on obciążenie wyjściowe. Twoja aplikacja może mieć zupełnie inne cechy, ale możesz zastosować podobne zasady do analizy obciążenia.

Oto zakończenie historyjki o Twitterze: po skutecznym zaimplementowaniu podejścia nr 2 Twitter przechodzi do hybrydowego rozwiązania łączącego obie techniki. Tweety większości użytkowników nadal są zapisywane w modelu rozgałęziania na osiach czasu w momencie publikowania wiadomości, przy czym nie dotyczy to niewielkiej grupy użytkowników (celebrytów) o bardzo dużej liczbie obserwatorów. Tweety celebrytów są pobierane niezależnie i dołączane do osi czasu obserwatora w trakcie jej wczytywania (tak jak w podejściu nr 1). To hybrydowe podejście zapewnia stabilnie wysoką wydajność. Do tego przykładu wrócisz w rozdziale 12. po zapoznaniu się z bardziej technicznymi informacjami.

## Opis wydajności

Po opisanu obciążenia możesz zbadać, co się stanie po jego wzroście. Możesz na to spojrzeć na dwa sposoby:

- Co się stanie z wydajnością systemu, jeśli obciążenie wzrośnie, a zasoby systemowe (procesor, pamięć, przepustowość sieci itd.) pozostaną niezmienione?
- O ile trzeba zwiększyć ilość zasobów, jeśli chcesz uzyskać tę samą wydajność po wzroście obciążenia?

Oba te pytania wymagają ustalenia liczb określających wydajność. Przyjrzyj się więc pokrótce temu, jak opisywać wydajność systemu.

W systemach przetwarzania wsadowego (np. Hadoop) zwykle ważna jest *przepustowość*, czyli liczba rekordów, jakie można przetwarzać w ciągu sekundy, lub łączny czas potrzebny do wykonania zadania na zbiorze danych o określonej wielkości<sup>2</sup>. W systemach internetowych zwykle ważniejszy jest *czas odpowiedzi* usługi, czyli czas od przesłania żądania przez klienta do otrzymania odpowiedzi.



### Opóźnienie a czas odpowiedzi

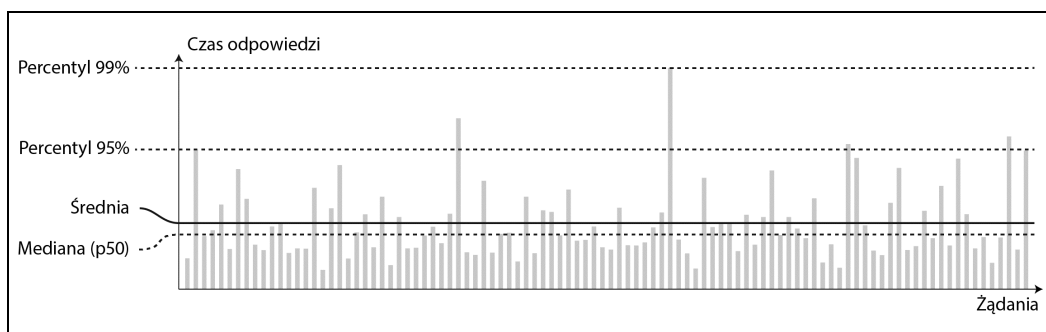
Określenia *opóźnienie* (ang. *latency*) i *czas odpowiedzi* (ang. *response time*) są często używane zamiennie, ale nie oznaczają one tego samego. Czas odpowiedzi mierzy się z perspektywy klienta. Obok czasu przetwarzania żądania (*czasu pracy usługi*) obejmuje też czas transferu w sieci i spędzony w kolejkach. Opóźnienie to czas oczekiwania żądania na obsłużenie. W tym okresie żądanie jest *uśpione* (oczekuje na usługę) [17].

<sup>2</sup> W idealnym świecie czas wykonywania zadania wsadowego to wielkość zbioru danych podzielona przez przepustowość. W praktyce ten czas jest często dłuższy. Jest to spowodowane niesymetrycznością (nierównym rozkładem danych między procesy robocze) i koniecznością oczekiwania na zakończenie pracy przez najwolniejsze zadanie.



Nawet jeśli wielokrotnie przesyłasz to samo żądanie, przy każdej próbie uzyskasz nieco odmienny czas odpowiedzi. W praktyce w systemie obsługującym różne żądania czas odpowiedzi może się znacznie wahać. Dlatego o czasie odpowiedzi trzeba myśleć nie jak o jednej liczbie, ale jak o *rozkładzie* wartości, który można zmierzyć.

Na rysunku 1.4 każdy szary słupek reprezentuje żądanie kierowane do usługi. Wysokość słupków pokazuje, ile czasu zajęła obsługa tych żądań. Większość żądań jest obsługiwanych szybko, jednak występują rzadkie *obserwacje odstające* o znacznie dłuższym czasie obsługi. Możliwe, że te wolno obsługiwane żądania są z natury bardziej kosztowne — np. wymagają przetworzenia większej ilości danych. Jednak nawet w sytuacji, gdy uważasz, że wszystkie żądania powinny zajmować tyle samo czasu, i tak pojawiają się różnice. Losowy dodatkowy czas może wynikać z przełączania kontekstu na proces tła, utraty pakietu sieciowego i retransmisji w protokole TCP, przerwy na odzyskiwanie pamięci, błędu stronicowania wymuszającego odczyt z dysku, mechanicznych wibracji szafy serwerowej [18] i wielu innych przyczyn.



Rysunek 1.4. Średnia i percentyle — czasy odpowiedzi dla próbki 100 żądań kierowanych do usługi

Często podaje się *średni* czas odpowiedzi usługi. Ściśle rzecz biorąc, określenie „średnia” nie dotyczy żadnego konkretnego wzoru. W praktyce zwykle rozumie się ją jako *średnią arytmetyczną* (jeśli danych jest  $n$  wartości, należy je zsumować, a następnie podzielić wynik przez  $n$ ). Jednak średnia nie jest bardzo dobrą miarą, gdy chcesz poznać „typowy” czas odpowiedzi, ponieważ nie pokazuje, ile użytkowników dotyczy dany czas odpowiedzi.

Zwykle lepiej posługiwać się *percentylami*. Jeśli przyjrzyysz się liście czasów odpowiedzi i posortujesz ją od najkrótszych do najwolniejszych, *mediana* to punkt środkowy. Na przykład jeśli mediana czasów odpowiedzi wynosi 200 ms, oznacza to, że połowa żądań jest przetwarzana w czasie krótszym niż 200 ms, a połowa żądań zajmuje więcej czasu.

Mediana to więc dobra miara, kiedy chcesz wiedzieć, jak długo użytkownicy zwykle muszą czekać. Połowa użytkowników jest obsługiwana w czasie krótszym niż mediana, a druga połowa w czasie dłuższym. Medianę nazywa się też *percentylem 50%*, co czasem skraca się do postaci *p50*. Zauważ, że mediana dotyczy jednego żądania. Jeśli dany użytkownik zgłasza kilka żądań (w trakcie sesji lub z powodu znajdowania się na jednej stronie wielu zasobów), prawdopodobieństwo, że przynajmniej jedno z nich będzie wymagało więcej czasu niż mediana, jest znacznie wyższe niż 50%.

Aby ustalić, jak kłopotliwe są obserwacje odstające, możesz się przyjrzeć wyższym percentyloom. Często wykorzystywane są percentyle 95%, 99% i 99,9% (*p95*, *p99* i *p999*). Są to wartości progowe czasów odpowiedzi, dla których 95%, 99% i 99,9% żądań jest przetwarzanych szybciej niż dana

wartość. Na przykład jeśli czas odpowiedzi dla percentyla 95% wynosi 1,5 s, oznacza to, że 95 ze 100 żądań jest przetwarzanych w czasie krótszym niż 1,5 s, a 5 ze 100 żądań wymaga 1,5 s lub więcej. Przedstawiono to na rysunku 1.4.

Wysokie percentyle czasów odpowiedzi, nazywane też *skrajnymi wartościami opóźnienia* (ang. *tail latencies*), są ważne, ponieważ bezpośrednio wpływają na komfort pracy użytkowników usługi. Na przykład Amazon opisuje wymogi dotyczące czasów odpowiedzi usług wewnętrznych w kategoriach percentyla 99,9%, choć czasy te dotyczą tylko 1 na 1000 żądań. Dzieje się tak, ponieważ klienci generujący najdłuższe żądania to często ci z największą ilością danych na kontach, co wynika z wielu zakupów. Oznacza to najcenniejszych klientów [19]. Ważne jest, aby dbać o zadowolenie tych klientów, zapewniając, że witryna szybko ich obsługuje. W Amazonie zaobserwowano też, że wydłużenie czasu odpowiedzi o 100 ms skutkuje spadkiem sprzedaży o 1% [20]. Z innych badań wynika, że jednosekundowe wydłużenie obsługi zmniejsza wskaźnik satysfakcji klientów o 16% [21, 22].

Optymalizowanie pod kątem percentyla 99,99% (najwolniejsze z 10 tys. żądań) okazało się jednak zbyt kosztowne i nie przyniosło wystarczających korzyści ze względu na cele Amazonu. Skracanie czasu odpowiedzi dla bardzo wysokich percentyli jest trudne, ponieważ wartości te są podatne na losowe zdarzenia pozostające poza naszą kontrolą, a korzyści są coraz mniejsze.

Percentyle są często używane w **SLO** (ang. *service level objective*) i **SLA** (ang. *service level agreement*). Są to kontrakty definiujące oczekiwaną wydajność i dostępność usługi. Kontrakt SLA może określać, że usługę uznaje się jako aktywną, jeśli mediana czasów odpowiedzi wynosi mniej niż 200 ms i percentyl 99% wynosi mniej niż 1 s (jeżeli czas odpowiedzi jest dłuższy, usługa równie dobrze mogłaby być wyłączona). Dodatkowym wymogiem może być to, by usługa była aktywna przez co najmniej 99,9% czasu. Te poziomy wyznaczają oczekiwania klientów usługi i pozwalają im domagać się zwrotu zapłaty, jeśli kontrakt SLA nie zostanie zrealizowany.

W przypadku wysokich percentyli za dużą część czasu odpowiedzi często odpowiadają kolejki. Ponieważ serwer potrafi jednocześnie przetwarzać tylko niewiele elementów (ogranicza to np. liczba rdzeni procesora), wystarczy mała liczba wolnych żądań, by wstrzymać przetwarzania dalszych żądań. To efekt, który można nazwać *blokowaniem na czole kolejki*. Nawet gdy serwer potrafi szybko przetworzyć te dalsze żądania, dla klienta czas odpowiedzi jest długi z powodu czasu oczekiwania na ukończenie wcześniejszych żądań. Z tego powodu ważne jest, by mierzyć czasy odpowiedzi po stronie klienta.

Gdy obciążenie jest generowane sztucznie w celu przetestowania skalowalności systemu, klient generujący obciążenie musi przysyłać żądania niezależnie od czasów odpowiedzi. Jeśli klient będzie oczekiwał na ukończenie wcześniejszego żądania przed przesłaniem następnego, w trakcie testów kolejki będą sztucznie krótsze niż w rzeczywistości, co zakłóca pomiary [23].

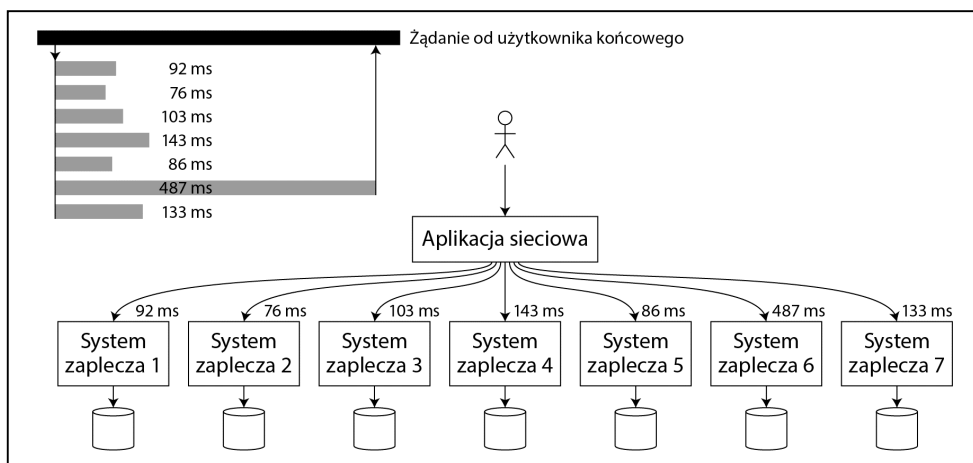
## Sposoby radzenia sobie z obciążeniem

Po omówieniu parametrów opisu obciążenia i wskaźników do pomiaru wydajności można na dobre rozpocząć analizę skalowalności — jak utrzymać wysoką wydajność, gdy obciążenie wzrośnie do określonego poziomu?

Architektura odpowiednia dla danego poziomu obciążenia zapewne nie poradzi sobie z jego 10-krotnym wzrostem. Dlatego jeśli pracujesz nad usługą, której obciążenie szybko rośnie, zapewne będziesz musiał na nowo przemyśleć architekturę po wzroście obciążenia o każdy rząd wielkości, a prawdopodobnie jeszcze częściej.

## Percentyle w praktyce

Wysokie percentyle są ważne zwłaszcza w usługach zaplecza wywoływanych wielokrotnie w ramach obsługi jednego żądania od użytkownika końcowego. Nawet jeśli te wywołania są zgłaszane równolegle, żądanie od użytkownika końcowego nadal musi oczekiwać na przetworzenie najwolniejszego z równoległych wywołań. Wystarczy jedno wolne wywołanie, aby obsługa całego żądania trwała długo. Przedstawia to rysunek 1.5. Nawet gdy tylko niewielki odsetek wywołań na zapleczu jest długo przetwarzany, ryzyko natrafienia na wolne wywołanie rośnie, jeśli żądanie użytkownika końcowego wymaga wielu wywołań. W takiej sytuacji także dla większego odsetka żądań użytkownika końcowego otrzymasz dłuższy czas odpowiedzi (ten efekt to *zwiększenie liczby skrajnych wartości opóźnienia* [24]).



Rysunek 1.5. Gdy obsłużenie żądania wymaga kilku wywołań do zaplecza, wystarczy jedno długie żądanie do zaplecza, aby wydłużyć przetwarzanie całego żądania od użytkownika końcowego

Jeśli chcesz dodać percentyle czasów odpowiedzi do panelu monitorowania usług, musisz na bieżąco wydajnie je obliczać. Możliwe, że zechcesz utrzymywać okno przesuwne czasów odpowiedzi na żądania z ostatnich 10 minut. Co minutę system może obliczać medianę i różne percentyle na podstawie wartości z tego okna i wyświetlać uzyskane dane na wykresie.

Naiwna implementacja polega na przechowywaniu listy czasów odpowiedzi dla wszystkich żądań z okna czasowego i sortowaniu tej listy co minutę. Jeśli to rozwiązanie jest dla Ciebie za mało wydajne, istnieją algorytmy, które potrafią obliczać dobre przybliżenie percentyli przy minimalnych kosztach procesora i pamięci. Są to algorytmy forward decay [25], t-digest [26] i HdrHistogram [27]. Pamiętaj, że uśrednianie percentyli, np. w celu zmniejszenia rozdzielczości czasowej lub połączenia danych z kilku maszyn, matematycznie nie ma sensu. Poprawną metodą agregowania danych z czasami odpowiedzi stanowi dodawanie histogramów [28].

Ludzie często mówią o rozbieżności między *skalowaniem pionowym* (ang. *scaling up* lub *vertical scaling*; polega to na wymianie maszyny na wydajniejszą) a *skalowaniem poziomym* (ang. *scaling out* lub *horizontal scaling*; jest to rozdzielanie obciążenia między wiele mniejszych maszyn). Rozdzielanie obciążenia między liczne maszyny nazywa się też architekturą *bez zasobów współdzielonych* (ang. *shared-nothing*). System, który potrafi działać na jednej maszynie, jest zwykle prostszy, jednak zaawansowane maszyny bywają bardzo drogie, dlatego przy bardzo wysokim obciążeniu roboczym często nie da się uniknąć skalowania poziomego. W praktyce skuteczne architektury to często pragmatyczne połączenie obu podejść. Na przykład zastosowanie kilku mocnych maszyn może się okazać prostsze i tańsze niż używanie dużej liczby małych maszyn wirtualnych.

Niektóre systemy są *elastyczne*, co oznacza, że potrafią automatycznie dodawać zasoby obliczeniowe po wykryciu wzrostu obciążenia. Inne są skalowane ręcznie (to człowiek analizuje zasoby i decyduje o dodaniu do systemu nowych maszyn). System elastyczny może być bardzo przydatny, gdy obciążenie jest wysoce nieprzewidywalne, jednak systemy skalowane ręcznie są prostsze i powodują mniej niespodzianek w trakcie eksploatacji (zob. punkt „Równoważenie partycji”).

Choć podział usług bezstanowych między wiele maszyn jest stosunkowo prosty, przenoszenie stanowych systemów danych z jednego węzła do środowiska rozproszonego może powodować wiele dodatkowej złożoności. Dlatego do niedawna powszechnie uważano, że bazę należy przechowywać w jednym węźle (ze skalowaniem pionowym) do czasu, gdy koszty skalowania lub wymogi wysokiej dostępności wymuszają przejście do środowiska rozproszonego.

Wraz ze wzrostem jakości narzędzi i abstrakcji z obszaru systemów rozproszonych to powszechne podejście może się zmienić (przynajmniej w niektórych aplikacjach). Możliwe, że w przyszłości rozproszone systemy danych staną się standardem — nawet w sytuacjach, gdy nie trzeba będzie obsługiwać dużych ilości danych lub dużego ruchu. Dalej w książce omówiono wiele rodzajów rozproszonych systemów danych. Dowiesz się nie tylko tego, w jakim stopniu są one skalowalne, ale też tego, jak łatwe jest ich użytkowanie i konserwowanie.

Architektura systemów działających na dużą skalę jest zwykle ściśle powiązana z aplikacją. Nie istnieje nic takiego jak uniwersalna skalowalna architektura na każdą okazję (nieformalnie można ją nazwać *magicznym sosem skalującym*). Problem może stanowić liczba odczytów, liczba zapisów, ilość przechowywanych danych, złożoność danych, wymogi dotyczące czasu odpowiedzi, wzorce dostępu lub (zazwyczaj) połączenie wszystkich tych i wielu innych kwestii.

Na przykład system zaprojektowany do obsługi 100 tys. żądań na sekundę (każde po 1 kB) wygląda zupełnie inaczej niż system zaprojektowany do obsługi trzech żądań na minutę (każde po 2 GB), nawet jeśli ilość danych w obu tych systemach jest podobna.

Architektura dobrze się skalująca w konkretnej aplikacji opiera się na założeniach dotyczących tego, które operacje będą często wykonywane, a które rzadko (są to parametry obciążenia). Jeśli te założenia okażą się błędne, praca inżynierska włożona w skalowanie w najlepszym razie zostanie zmarnowana, a w najgorszym przyniesie skutki odwrotne od zamierzonych. W startupach na wczesnym etapie rozwoju lub w nieprzetestowanych produktach zwykle ważniejsza jest możliwość szybkiego dodawania funkcji produktu niż myślenie o skalowaniu pod kątem hipotetycznego przyszłego obciążenia.

Nawet gdy skalowalną architekturę dostosowuje się do konkretnej aplikacji, zwykle jest zbudowana z komponentów o ogólnym przeznaczeniu łączonych w znane wzorce. W tej książce opisano takie komponenty i wzorce.

## Łatwość konserwacji

Wiadomo, że większość kosztów związanych z oprogramowaniem jest ponoszonych nie w trakcie jego budowania, ale w czasie bieżącej eksploatacji — w związku z naprawianiem błędów, utrzymywaniem zdolności operacyjnej systemów, badaniem awarii, dostosowywaniem do nowych platform, modyfikowaniem na potrzeby nowych przypadków użycia, spłacaniem długu technicznego i dodawaniem nowych funkcji.

Jednak, niestety, wiele osób pracujących nad systemami oprogramowania nie lubi konserwacji *przestarzałych* (ang. *legacy*) systemów. Możliwe, że taka konserwacja wymaga naprawiania cudzych pomyłek albo pracy z nieaktualnymi platformami lub systemami zmuszonymi wykonywać zadania, do których robienia nie są przeznaczone. Każdy przestarzały system jest kłopotliwy w inny sposób. Dlatego tak trudno przedstawić ogólne wskazówki.

Jednak można i należy projektować oprogramowanie tak, aby starać się zminimalizować trudności w trakcie konserwacji i samemu uniknąć tworzenia przestarzałego oprogramowania. W tym celu warto zwrócić uwagę przede wszystkim na trzy zasady projektowania systemów informatycznych:

### *Łatwość eksploatacji*

Należy umożliwić pracownikom operacyjnym łatwe zapewnianie płynnej pracy systemu.

### *Prostota*

Nowi inżynierowie powinni móc łatwo zrozumieć system. W tym celu należy w maksymalnym stopniu wyeliminować z niego złożoność. Zauważ, że nie jest to tym samym co prostota interfejsu użytkownika.

### *Łatwość modyfikowania*

Zadbaj o to, by inżynierowie mogli w przyszłości łatwo wprowadzać zmiany w systemie, dostosowując go do nieprzewidzianych przypadków użycia po zmianie wymagań. Tę cechę można też nazwać *rozszerzalnością*, *modyfikowalnością* lub *plastycznością*.

Podobnie jak w kontekście niezawodności i skalowalności, nie istnieją łatwe rozwiązania pozwalające osiągać te cele. Zamiast tego warto spróbować zastanowić się nad systemami z uwzględnieniem łatwości eksploatacji, prostoty i łatwości modyfikowania.

## Łatwość eksploatacji — ułatwianie życia pracownikom operacyjnym

Pojawiają się sugestie, że „dobry zespół operacyjny często potrafi poradzić sobie z ograniczeniami kiepskiego (lub niekompletnego) oprogramowania, jednak dobre oprogramowanie nie będzie działać stabilnie przy kiepskim zespole operacyjnym” [12]. Choć niektóre aspekty eksploatacji można i należy automatyzować, to ludzie odpowiadają za konfigurowanie automatyzacji i zapewnianie jej poprawnego działania.

Zespół operacyjny jest niezbędny do zapewniania płynnego działania systemów informatycznych. Dobry zespół operacyjny zwykle odpowiada za czynniki wymienione poniżej i inne [29]:

- Monitorowanie stanu systemu i szybkie przywracanie usługi, jeśli znajdzie się w nieodpowiednim stanie.
- Wykrywanie przyczyn problemów takich jak awarie systemu lub spadek wydajności.
- Zapewnianie aktualności oprogramowania i platform (w tym: instalowanie poprawek bezpieczeństwa).
- Badanie tego, jak różne systemy wpływają na siebie, co pozwala uniknąć wprowadzania mogących rodzić problemy zmian, zanim spowodują szkody.
- Przewidywanie przyszłych problemów i rozwiązywanie ich przed wystąpieniem (planowanie wydajności).
- Opracowywanie dobrych praktyk i narzędzi wdrażania oprogramowania, zarządzania konfiguracją itd.
- Wykonywanie złożonych zadań z zakresu eksploatacji (np. przenoszenie aplikacji z jednej platformy na inną).
- Utrzymywanie bezpieczeństwa systemu w trakcie wprowadzania zmian w konfiguracji.
- Definiowanie procesów, dzięki którym eksploatacja jest przewidywalna, i pomaganie w utrzymywaniu stabilności środowiska produkcyjnego.
- Zachowywanie w organizacji wiedzy na temat systemu, nawet gdy poszczególne osoby przychodzą do firmy i z niej odchodzą.

Łatwość eksploatacji oznacza, że wykonywanie rutynowych zadań jest proste, dzięki czemu zespół operacyjny może się skoncentrować na działaniach o wysokiej wartości. Systemy danych mogą na różne sposoby ułatwiać wykonywanie rutynowych prac. Oto niektóre z tych sposobów:

- Zapewnianie wglądu w pracę środowiska uruchomieniowego i wewnętrzne mechanizmy systemu za pomocą dobrego systemu monitorowania.
- Zapewnianie dobrej obsługi automatyzacji i integracji ze standardowymi narzędziami.
- Unikanie zależności od pojedynczych maszyn (pozwała to na wyłączenie maszyny na potrzeby konserwacji, a cały system działa wtedy bez zakłóceń).
- Zapewnianie dobrej dokumentacji i łatwego do zrozumienia modelu operacyjnego („Jeśli zrobię X, nastąpi Y”).
- Zapewnianie skutecznego działania domyślnego, ale też dawanie administratorom swobody do zmiany w razie potrzeby wprowadzenia ustawień domyślnych.
- Umożliwianie automatycznej naprawy tam, gdzie to wskazane, i zapewnianie administratorom ręcznej kontroli nad stanem systemu, jeśli to konieczne.
- Zapewnianie przewidywalnej pracy i minimalizowanie niespodzianek.

## Prostota — zarządzanie złożonością

W niewielkich projektach informatycznych kod może być cudownie prosty i zrozumiały. Jednak gdy projekt się rozrasta, kod często staje się bardzo skomplikowany i trudny do analizy. Ta złożoność spowalnia wszystkich, którzy muszą pracować nad systemem, co dodatkowo zwiększa koszty konserwacji. Projekt informatyczny pogrążony w złożoności można nazwać *wielką bryłą błota* [30].

Istnieją różne objawy złożoności: znaczny wzrost ilości miejsca potrzebnego na stan, ściśle powiązanie między modułami, skomplikowane zależności, niespójne nazewnictwo i terminologia, sztuczki służące rozwiązaniu problemów z wydajnością, specjalne przypadki rozwiązujące problemy z innych miejsc itd. Wiele na ten temat już napisano [31, 32, 33].

Gdy złożoność utrudnia konserwację, często następuje przekroczenie budżetów i harmonogramów. W skomplikowanym oprogramowaniu rośnie też ryzyko pojawienia się błędów w trakcie wprowadzania zmian. Gdy system dla programistów jest trudny do zrozumienia i analizy, łatwiej może być przeoczyć ukryte założenia, niezamierzone konsekwencje i nieoczekiwane interakcje. Natomiast ograniczenie złożoności znacznie ułatwia konserwację oprogramowania. Dlatego prostota powinna być ważnym celem w budowanych systemach.

Upraszczanie systemu nie zawsze oznacza ograniczenia jego funkcji. Może też polegać na eliminowaniu *przypadkowej* złożoności. Moseley i Marks [32] definiują złożoność jako przypadkową, jeśli nie jest nieodłączna od problemu rozwiązywanego przez oprogramowanie (z perspektywy użytkownika), a powstaje tylko z powodu implementacji.

Jedno z najlepszych narzędzi do eliminowania przypadkowej złożoności stanowi *abstrakcja*. Dobra abstrakcja pozwala ukryć wiele szczegółów implementacji za przejrzystą i łatwą do zrozumienia fasadą. Dobrą abstrakcję można też wykorzystać w wielu różnych aplikacjach. Takie ponowne wykorzystanie kodu jest nie tylko wydajniejsze od wielokrotnego pisania podobnego rozwiązania, ale też prowadzi do powstawania oprogramowania wyższej jakości, ponieważ wzrost poziomu wyodrębnionego komponentu przynosi korzyść we wszystkich aplikacjach, którego go używają.

Na przykład wysokopoziomowe języki programowania są abstrakcjami, które ukrywają kod maszynowy, rejestry procesora i wywołania systemowe. SQL to abstrakcja ukrywająca złożone struktury danych używane na dysku i w pamięci, równoległe żądania od innych klientów i niespójności występujące po awariach. Oczywiście w trakcie programowania w języku wysokopoziomowym nadal posługujesz się kodem maszynowym, ale nie robisz tego *bezpośrednio*, ponieważ abstrakcja w postaci języka programowania sprawia, że nie musisz myśleć o takim kodzie.

Jednak znajdowanie wartościowych abstrakcji jest bardzo trudne. Choć w obszarze systemów rozproszonych istnieje wiele dobrych algorytmów, nie ma jasności, jak łączyć je w abstrakcje pomagające utrzymać złożoność systemu na akceptowalnym poziomie.

W tej książce zwracam uwagę na przydatne abstrakcje, które umożliwiają wyodrębnienie fragmentów dużych systemów i przekształcenie ich w dobrze zdefiniowane komponenty wielokrotnego użytku.

## Łatwość modyfikowania — umożliwianie prostego wprowadzania zmian

Jest bardzo mało prawdopodobne, że wymagania stawiane Twojemu systemowi pozostaną na zawsze niezmiennie. Z dużo większym prawdopodobieństwem będą się ciągle zmieniać. Poznasz nowe fakty, pojawią się nieprzewidziane wcześniej przypadki użycia, zmienią się priorytety biznesowe, użytkownicy będą żądać nowych funkcji, starsze platformy zostaną zastąpione nowymi, zmienią się wymogi prawne lub ustawowe, rozrastanie się systemu wymusi zmiany architektoniczne itd.

W obszarze procesów organizacyjnych model dostosowywania się do zmian opisano w postaci wzorców *zwinnych* (ang. *agile*). Społeczność skupiona wokół metod zwinnych opracowała też narzędzia techniczne i wzorce pomocne w rozwijaniu oprogramowania w często zmieniającym się środowisku. Te narzędzia to np. programowanie sterowane testami (ang. *test-driven development* — **TDD**) i refaktoryzacja.

Większość technik zwinnych działa w stosunkowo niewielkiej, lokalnej skali (np. kilku plików z kodem źródłowym jednej aplikacji). W tej książce szukam sposobów zwiększenia zwinności na poziomie większego systemu danych, który może obejmować kilka różnych aplikacji lub usług o rozmaitych cechach. Jak np. przeprowadzić „refaktoryzację” architektury tworzenia osi czasu na Twitterze (zob. punkt „Opisywanie obciążenia”), aby przejść od podejścia nr 1 do podejścia nr 2?

Łatwość modyfikowania systemu danych i dostosowywania go do zmieniających się wymagań jest ściśle powiązana z prostotą i abstrakcjami. Proste i łatwe do zrozumienia systemy są zwykle łatwiejsze do modyfikowania niż złożone. Jednak ponieważ jest to tak ważne zagadnienie, do opisu zwinności na poziomie systemu danych używa się innego słowa — *łatwość modyfikowania* (ang. *evolvability*) [34].

## Podsumowanie

W tym rozdziale opisano podstawowe sposoby myślenia o aplikacjach intensywnie przetwarzających dane. Te zasady posłużą za wskazówki w dalszych rozdziałach w trakcie omawiania szczegółów technicznych.

Aplikacja musi spełniać różne wymagania, aby była przydatna. Istnieją *wymagania funkcjonalne* (określające, co aplikacja powinna robić — np. umożliwiać przechowywanie, pobieranie, wyszukiwanie i przetwarzanie danych na różne sposoby), a także *wymagania niefunkcjonalne* (są to ogólne cechy takie jak bezpieczeństwo, niezawodność, zgodność z prawem, skalowalność, kompatybilność i łatwość konserwacji). W tym rozdziale szczegółowo omówiono niezawodność, skalowalność i łatwość konserwacji.

*Niezawodność* oznacza, że system działa prawidłowo nawet po wystąpieniu błędów. Źródłem błędów może być sprzęt (te kłopoty są przeważnie losowe i nieskorelowane), oprogramowanie (wtedy problemy są zwykle powtarzalne i trudno sobie z nimi poradzić) oraz ludzie (którzy, co nieuniknione, od czasu do czasu popełniają pomyłki). Techniki zapewniania odporności na błędy pozwalają ukryć niektóre rodzaje usterek przed użytkownikami końcowymi.

*Skalowalność* oznacza stosowanie strategii utrzymywania wysokiej wydajności nawet w obliczu wzrostu obciążenia. Aby omówić skalowalność, najpierw potrzebne są metody ilościowego opisu obciążenia i wydajności. W ramach przykładu opisywania obciążenia pokrótce przedstawiano os



czasu z Twittera. Jako przykład sposobu pomiaru wydajności posłużyły percentyle czasów odpowiedzi. W skalowalnym systemie można dodać moc obliczeniową, aby system pozostał niezawodny przy wysokim obciążeniu.

*Łatwość konserwacji* ma wiele aspektów, jednak istotę tego zagadnienia stanowi ułatwianie pracy inżynierom i zespołom operacyjnym pracującym z danym systemem. Wartościowe abstrakcje pomagają ograniczyć złożoność i sprawiają, że system łatwiej jest modyfikować i dostosowywać do nowych przypadków użycia. Wysoka jakość operacyjna wymaga dobrego wglądu w stan systemu i skutecznych sposobów zarządzania tym stanem.

Niestety, nie istnieją łatwe metody zapewniania niezawodności, skalowalności lub łatwości konserwacji. Są jednak wzorce i techniki, które często się pojawiają w aplikacjach różnego rodzaju. W kilku następnych rozdziałach przyjrzymy się przykładowym systemom danych i analizie tego, jak osiągać opisane tu cele.

Dalej, w części III, poznasz wzorce tworzenia systemów składających się z kilku współpracujących ze sobą komponentów. Są to systemy podobne do tego z rysunku 1.1.

## **Literatura cytowana**

- [1] Michael Stonebraker i Uğur Çetintemel, „One Size Fits All”: *An Idea Whose Time Has Come and Gone*, z: *21st International Conference on Data Engineering (ICDE)*, kwiecień 2005 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf>).
- [2] Walter L. Heimerdinger i Charles B. Weinstock, *A Conceptual Framework for System Fault Tolerance*, Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, październik 1992 (<https://www.sei.cmu.edu/reports/92tr033.pdf>).
- [3] Ding Yuan, Yu Luo, Xin Zhuang i in., *Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems*, z: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, październik 2014 (<https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf>).
- [4] Yuri Izrailevsky i Ariel Tseitlin, *The Netflix Simian Army*, techblog.netflix.com, 19 lipca 2011 (<https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>).
- [5] Daniel Ford, François Labelle, Florentina I. Popovici i in., *Availability in Globally Distributed Storage Systems*, z: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, październik 2010 (<https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/36737.pdf>).
- [6] Brian Beach, *Hard Drive Reliability Update — Sep 2014*, backblaze.com, 23 września 2014 (<https://www.backblaze.com/blog/hard-drive-reliability-update-september-2014/>).
- [7] Laurie Voss, *AWS: The Good, the Bad and the Ugly*, blog.awe.sm, 18 grudnia 2012 (<https://web.archive.org/web/20160429075023/http://blog.awe.sm/2012/12/18/aws-the-good-the-bad-and-the-ugly/>).

- [8] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa i in., *What Bugs Live in the Cloud?*, z: *5th ACM Symposium on Cloud Computing (SoCC)*, listopad 2014 (<http://ucare.cs.uchicago.edu/pdf/socc14-cbs.pdf>; <https://dl.acm.org/citation.cfm?doid=2670979.2670986>).
- [9] Nelson Minar, *Leap Second Crashes Half the Internet*, *somebits.com*, 3 lipca 2012 (<http://www.somebits.com/weblog/tech/bad/leap-second-2012.html>).
- [10] Amazon Web Services, *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*, *aws.amazon.com*, 29 kwietnia 2011 (<https://aws.amazon.com/message/65648/>).
- [11] Richard I. Cook, *How Complex Systems Fail*, Cognitive Technologies Laboratory, kwiecień 2000 (<http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf>).
- [12] Jay Kreps, *Getting Real About Distributed System Reliability*, *blog.empathy-box.com*, 19 marca 2012 (<http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability>).
- [13] David Oppenheimer, Archana Ganapathi i David A. Patterson, *Why Do Internet Services Fail, and What Can Be Done About It?*, z: *4th USENIX Symposium on Internet Technologies and Systems (USITS)*, marzec 2003 ([http://static.usenix.org/legacy/events/usits03/tech/full\\_papers/oppenheimer/oppenheimer.pdf](http://static.usenix.org/legacy/events/usits03/tech/full_papers/oppenheimer/oppenheimer.pdf)).
- [14] Nathan Marz, *Principles of Software Engineering, Part 1*, *nathanmarz.com*, 2 kwietnia 2013 (<http://nathanmarz.com/blog/principles-of-software-engineering-part-1.html>).
- [15] Michael Jurewitz, *The Human Impact of Bugs*, *jury.me*, 15 marca 2013 (<http://jury.me/blog/2013/3/14/the-human-impact-of-bugs>).
- [16] Raffi Krikorian, *Timelines at Scale*, z: *QCon San Francisco*, listopad 2012.
- [17] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002, ISBN: 978-0-321-12742-6.
- [18] Kelly Sommers, *After all that run around, what caused 500ms disk latency even when we replaced physical server?*, *twitter.com*, 13 listopada 2014 (<https://twitter.com/kellabyte/status/532930540777635840>).
- [19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani i in., *Dynamo: Amazon's Highly Available Key-Value Store*, z: *21st ACM Symposium on Operating Systems Principles (SOSP)*, październik 2007 (<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>).
- [20] Greg Linden, *Make Data Useful*, slajdy z prezentacji z kursu eksploracji danych na Uniwersytecie Stanforda (CS345), grudzień 2006 (<http://glinden.blogspot.co.uk/2006/12/slides-from-my-talk-at-stanford.html>).
- [21] Tammy Everts, *The Real Cost of Slow Time vs Downtime*, *webperformancetoday.com*, 12 listopada 2014 (<https://blog.radware.com/applicationdelivery/wpo/2014/11/real-cost-slow-time-vs-downtime-slides/>).

- [22] Jake Brutlag, *Speed Matters for Google Web Search*, [googleresearch.blogspot.co.uk](http://googleresearch.blogspot.co.uk), 22 czerwca 2009 (<https://research.googleblog.com/2009/06/speed-matters.html>).
- [23] Tyler Treat, *Everything You Know About Latency Is Wrong*, [bravenewgeek.com](http://bravenewgeek.com), 12 grudnia 2015 (<http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/>).
- [24] Jeffrey Dean i Luiz André Barroso, *The Tail at Scale*, „Communications of the ACM”, rocznik 56, nr 2, s. 74 – 80, luty 2013 (<https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/abstract>; <https://dl.acm.org/citation.cfm?doid=2408776.2408794>).
- [25] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava i Bojian Xu, *Forward Decay: A Practical Time Decay Model for Streaming Systems*, z: *25th IEEE International Conference on Data Engineering (ICDE)*, marzec 2009 (<http://dimacs.rutgers.edu/~graham/pubs/papers/fwddecay.pdf>).
- [26] Ted Dunning i Otmar Ertl, *Computing Extremely Accurate Quantiles Using t-Digests*, [github.com](https://github.com), marzec 2014 (<https://github.com/tdunning/t-digest>).
- [27] Gil Tene, *HdrHistogram*, [hdrhistogram.org](http://www.hdrhistogram.org) (<http://www.hdrhistogram.org/>).
- [28] Baron Schwartz, *Why Percentiles Don't Work the Way You Think*, [vividcortex.com](http://www.vividcortex.com), 7 grudnia 2015 (<https://www.vividcortex.com/blog/why-percentiles-dont-work-the-way-you-think>).
- [29] James Hamilton, *On Designing and Deploying Internet-Scale Services*, z: *21st Large Installation System Administration Conference (LISA)*, listopad 2007 ([https://www.usenix.org/legacy/events/lisa07/tech/full\\_papers/hamilton/hamilton.pdf](https://www.usenix.org/legacy/events/lisa07/tech/full_papers/hamilton/hamilton.pdf)).
- [30] Brian Foote i Joseph Yoder, *Big Ball of Mud*, z: *4th Conference on Pattern Languages of Programs (PLoP)*, wrzesień 1997 (<http://www.laputan.org/pub/foote/mud.pdf>).
- [31] Frederick P. Brooks, *No Silver Bullet — Essence and Accident in Software Engineering*, z: „The Mythical Man-Month”, wydanie rocznicowe, Addison-Wesley, 1995, ISBN: 978-0-201-83595-3.
- [32] Ben Moseley i Peter Marks, *Out of the Tar Pit*, z: „BCS Software Practice Advancement” (SPA), 2006 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8928>).
- [33] Rich Hickey, *Simple Made Easy*, z: „Strange Loop”, wrzesień 2011 (<https://www.infoq.com/presentations/Simple-Made-Easy>).
- [34] Hongyu Pei Breivold, Ivica Crnkovic i Peter J. Eriksson, *Analyzing Software Evolvability*, z: *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, lipiec 2008 (<http://www.mrtc.mdh.se/publications/1478.pdf>; <http://ieeexplore.ieee.org/document/4591576/>).



# Modele danych i języki zapytań

*Granice naszego języka wyznaczają granice naszego świata.*

— Ludwig Wittgenstein, *Traktat logiczno-filozoficzny* (1922)

Modele danych są prawdopodobnie najważniejszym aspektem tworzenia oprogramowania, ponieważ mają bardzo duży wpływ nie tylko na to, jak oprogramowanie jest pisane, ale też na to, jak *myślimy o problemie*, który rozwiązujemy.

Większość aplikacji tworzy się z użyciem położonych jedna nad drugą warstw modeli danych. W kontekście każdej warstwy najważniejsze pytanie brzmi: jak jest ona *reprezentowana* w kategoriach następnej niższej warstwy? Oto przykład:

1. Autor aplikacji analizuje rzeczywisty świat (w którym występują ludzie, organizacje, towary, działania, przepływ pieniędzy, czujniki itd.) i modeluje go za pomocą obiektów lub struktur danych oraz interfejsów API operujących na tych strukturach. Te struktury często są specyficzne dla aplikacji.
2. Te struktury danych na potrzeby ich przechowywania są przedstawiane za pomocą modelu danych ogólnego przeznaczenia — np. dokumentów JSON lub XML, tabel w bazie relacyjnej lub modelu opartego na grafie.
3. Inżynierowie, którzy zbudowali oprogramowanie bazy danych, ustalili, jak dane w formacie JSON lub XML, z bazy relacyjnej albo z grafu są reprezentowane jako bajty w pamięci, na dysku lub w sieci. Ta reprezentacja może pozwalać na kierowanie zapytań o dane, przeszukiwanie ich, operowanie na nich i przetwarzanie na różne sposoby.
4. Na jeszcze niższym poziomie inżynierowie sprzętu określili, jak reprezentować bajty w kategoriach prądu, impulsów światła, pola magnetycznego itd.

W złożonej aplikacji może występować więcej poziomów pośrednich, np. interfejsy API oparte na innych takich interfejsach, jednak podstawowa zasada jest stała — każda warstwa ukrywa złożoność niższych warstw, udostępniając przejrzysty model danych. Poszczególne abstrakcje umożliwiają skuteczną współpracę różnym grupom ludzi, np. inżynierom producenta bazy danych i twórcom aplikacji używającym tej bazy.

Istnieje wiele różnych rodzajów modeli danych, a w każdym modelu uwzględnione są założenia na temat jego używania. Niektóre sposoby użytkowania są proste, a inne nie są obsługiwane. Część operacji działa szybko, inne są mało wydajne. Niektóre transformacje danych przebiegają naturalnie, inne są kłopotliwe.

Opanowanie choćby jednego modelu danych może wymagać dużo wysiłku (pomyśl o tym, ile książek poświęcono relacyjnemu modelowaniu danych). Tworzenie oprogramowania jest trudne nawet wtedy, gdy pracujesz tylko z jednym modelem danych i nie musisz się martwić jego wewnętrznymi mechanizmami. Jednak ponieważ model danych ma tak duży wpływ na to, co używające go oprogramowanie może zrobić, a czego nie da rady, istotne jest, aby dobrać model odpowiedni dla aplikacji.

W tym rozdziale poznasz kilka modeli danych ogólnego przeznaczenia służących do składowania danych i kierowania do nich zapytań (punkt 2. na przedstawionej wcześniej liście). Znajdziesz tu porównanie modelu relacyjnego, modelu opartego na dokumentach i kilku modeli opartych na grafach. Opisane są też różne języki zapytań wraz z porównaniem przypadków użycia. Rozdział 3. zawiera omówienie działania systemów składowania danych, czyli tego, jak opisywane modele są zaimplementowane (punkt 3. z listy).

## Model relacyjny a model oparty na dokumentach

Obecnie prawdopodobnie najbardziej popularny model danych to ten używany w SQL-u i oparty na modelu relacyjnym zaproponowanym w 1970 r. przez Edgara Codd'a [1]. Dane są w nim uporządkowane w *relacje* (nazywane w SQL-u *tabelami*), a każda relacja to nieuporządkowana kolekcja *krotek* (*wierszy* w SQL-u).

Model relacyjny był propozycją teoretyczną, początkowo wiele osób wątpiło, że można go wydajnie zaimplementować. Jednak do połowy lat 80. systemy **RDBMS** (ang. *relational database management systems*, czyli systemy zarządzania relacyjnymi bazami danych) i SQL stały się narzędziami stosowanymi z wyboru przez większość osób, które potrzebowały przechowywać dane w regularnej strukturze i zgłaszać zapytania o nie. Dominacja relacyjnych baz danych trwała ok. 25 – 30 lat, co w historii informatyki stanowi wieczność.

Korzenie relacyjnych baz danych sięgają *przetwarzania danych biznesowych* wykonywanego na komputerach typu mainframe w latach 60. i 70. ubiegłego wieku. Z dzisiejszej perspektywy ówczesne zastosowania baz wydają się nudne. Było to głównie *przetwarzanie transakcji* (wprowadzanie transakcji sprzedaży lub bankowych, rezerwacje biletów lotniczych, śledzenie stanu magazynu) i *przetwarzanie wsadowe* (obsługa faktur, list płac, raportów).

Wcześniejsze bazy zmuszały programistów aplikacji do długiego zastanawiania się nad wewnętrzną reprezentacją danych w bazie. Celem modelu relacyjnego było ukrycie tego szczegółu implementacji za prostszym interfejsem.

Przez lata pojawiło się wiele konkurencyjnych metod składowania danych i zgłaszania zapytań. W latach 70. i na początku lat 80. głównymi alternatywnymi rozwiązaniami były *model sieciowy* i *model hierarchiczny*, jednak model relacyjny je wyparł. Bazy obiektowe pojawiły się na krótki czas pod koniec lat 80. i na początku lat 90. Z początkiem naszego wieku pojawiły się bazy XML-owe, jednak nie zdobyły popularności. Wszystkim konkurentom modelu relacyjnego towarzyszyło wiele rozgłosu, ale żadne z tych rozwiązań nie przetrwało [2].

Gdy komputery stały się znacznie wydajniejsze i zyskały połączenie z siecią, zaczęto ich używać w coraz bardziej różnorodnych celach. Niezwykle jest to, że bazy relacyjne można było bardzo skutecznie uogólnić i — obok pierwotnego przeznaczenia, czyli przetwarzania danych biznesowych — wykorzy-

stać w różnorodnych zastosowaniach. Wiele rozwiązań spotykanych obecnie w internecie nadal wykorzystuje bazy relacyjne. Takie bazy są używane np. w publikacjach internetowych, dyskusjach, sieciach społecznościowych, handlu elektronicznym, grach, aplikacjach biurowych typu SaaS i wielu innych obszarach.

## Powstanie modelu NoSQL

Obecnie, w drugiej dekadzie naszego wieku, NoSQL stanowi najnowszą próbę zerwania z dominacją modelu relacyjnego. Nazwa NoSQL jest niefortunna, ponieważ nie określa żadnej konkretnej technologii. Początkowo, w 2009 r., miała być tylko chwytliwym tagiem na Twitterze dotyczącym spotkań poświęconych otwartym, rozproszonym, nierelacyjnym bazom [3]. Mimo to ta nazwa się przyjęła i szybko zyskała popularność w społeczności startupów internetowych i nie tylko. Obecnie wiele ciekawych systemów bazodanowych jest łączonych z tagiem #NoSQL, a znaczenie tej nazwy zinterpretowano później jako *Not Only SQL* [4].

Wzrost popularności baz NoSQL wynika z kilku czynników. Oto niektóre z nich:

- Potrzeba większej skalowalności niż ta, jaką można łatwo uzyskać za pomocą baz relacyjnych (wiąże się to m.in. z bardzo dużymi zbiorami danych i bardzo wysoką liczbą operacji zapisu).
- Powszechne preferowanie bezpłatnego i otwartego oprogramowania zamiast komercyjnych produktów bazodanowych.
- Wyspecjalizowane operacje w zapytaniach, słabo obsługiwane w modelu relacyjnym.
- Frustracja spowodowana ograniczonością schematów relacyjnych oraz potrzeba tworzenia bardziej dynamicznego i dającego większe możliwości modelu danych [5].

W różnych aplikacjach występują różne wymagania, a w poszczególnych przypadkach użycia optymalne mogą być inne technologie. Dlatego możliwe, że w przewidywalnej przyszłości bazy relacyjne będą używane równolegle z różnorodnymi nierelacyjnymi magazynami danych. To rozwiązanie czasem nazywa się *polyglot persistence* [3].

## Niedopasowanie modeli obiektowego i relacyjnego

Obecnie większość aplikacji jest pisanych w językach obiektowych, co prowadzi do często przytaczanego zarzutu wobec SQL-owego modelu danych. Jeśli dane są przechowywane w tabelach relacyjnych, potrzebna jest kłopotliwa warstwa odpowiedzialna za przekształcenia między obiektami z kodu aplikacji a modelem tabel, wierszy i kolumn z bazy danych. Niespójność między tymi modelami czasem nazywa się *niezgodnością impedancji*<sup>1</sup>.

Platformy **ORM** (ang. *object-relational mapping*) takie jak ActiveRecord i Hibernate ograniczają ilość szablonowego kodu potrzebnego w warstwie przekształceń, jednak nie potrafią całkowicie ukryć różnic między oboma modelami.

---

<sup>1</sup> Jest to pojęcie zapożyczone z elektroniki. Każdy obwód elektryczny ma określoną impedancję (opór w obwodach prądu zmiennego) wejść i wyjść. Gdy podłączasz wyjście jednego obwodu do wejścia innego obwodu, przepływ mocy przez połączenie jest maksymalny, jeśli impedancje wyjścia i wejścia są ze sobą zgodne. Niezgodność impedancji może prowadzić do odbijania sygnału i innych problemów.

Na przykład na rysunku 2.1 pokazano, jak życiorys (profil w serwisie LinkedIn) może wyglądać w schemacie relacyjnym. Cały profil można zidentyfikować na podstawie unikatowego identyfikatora `user_id`. Pola takie jak `first_name` i `last_name` występują raz na użytkownika, dlatego można je przedstawić jako kolumny tabeli `users`. Jednak większość osób wykonuje w swojej karierze więcej niż jedną pracę (zajmuje różne stanowiska). Ludzie mają też różne poziomy edukacji i liczbę kontaktów. Między użytkownikiem a tymi elementami występuje relacja jeden do wielu, którą można przedstawić na różne sposoby:

- W tradycyjnym modelu SQL-owym (przed wersją SQL:1999) najczęściej spotykana znormalizowana reprezentacja obejmowałaby stanowiska, wykształcenie i dane kontaktowe w odrębnych tabelach z referencjami w postaci klucza obcego do tabeli `users` (tak jak na rysunku 2.1).
- W późniejszych wersjach SQL-a dodano obsługę ustrukturyzowanych typów danych i XML-a. To umożliwiło przechowywanie danych z wieloma wartościami w jednym wierszu z możliwością zgłaszania zapytań i indeksowania takich dokumentów. Te możliwości są dostępne w różnym stopniu w systemach Oracle, IBM DB2, MS SQL Server i PostgreSQL [6, 7]. W kilku bazach obsługiwane są też dane w formacie JSON (są to np. bazy IBM DB2, MySQL i PostgreSQL) [8].
- Trzecia możliwość to zapisanie stanowisk, wykształcenia i danych kontaktowych jako dokumentu w formacie JSON lub XML, umieszczenie go w kolumnie tekstowej w bazie i umożliwienie aplikacji interpretowania struktury i zawartości tych dokumentów. W tej konfiguracji zwykle nie można kierować do bazy zapytań o wartości z tak zapisanej kolumny.

Dla struktury danych odpowiadającej życiorysowi, który stanowi w zasadzie niezależny *dokument*, reprezentacja w formacie JSON może być odpowiednia (zob. listing 2.1). Zaleta formatu JSON polega na tym, że jest znacznie prostszy niż XML. Ten model danych obsługuje się w bazach opartych na dokumentach, np. w MongoDB [9], RethinkDB [10], CouchDB [11] i Espresso [12].

Listing 2.1. Reprezentowanie profilu z serwisu LinkedIn jako dokumentu w formacie JSON

```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Współprzewodniczący fundacji Billa i Melindy Gatesów... Aktywny blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Współprzewodniczący", "organization": "Fundacja Billa i Melindy Gatesów"},
    {"job_title": "Współzałożyciel, prezes", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```



<http://www.linkedin.com/in/williamhgates>



## Bill Gates

Obszar metropolitalny  
Seattle | Filantropia

### Podsumowanie

Współprzewodniczący fundacji Billa i Melindy Gatesów. Prezes Microsoft Corporation. Zagorzały czytelnik. Zapalony podróżnik. Aktywny blogger.

### Doświadczenie

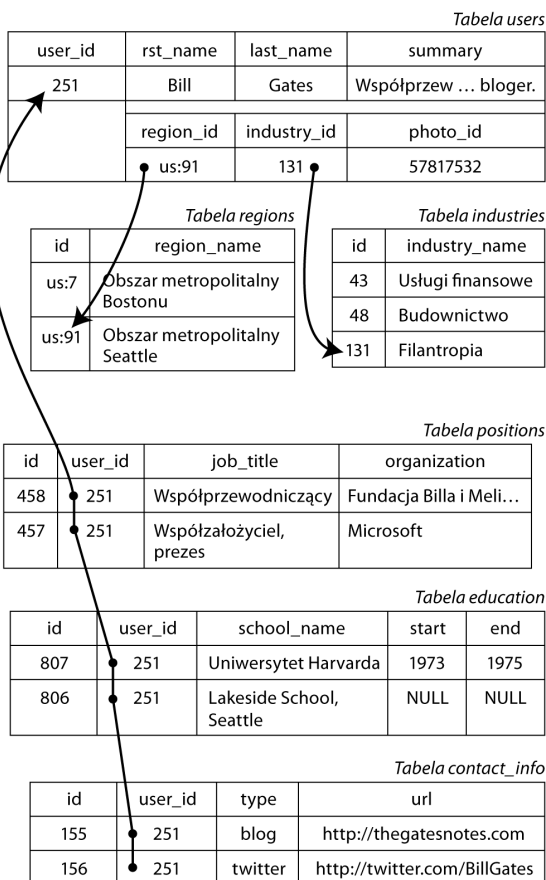
Współprzewodniczący • Fundacja Billa i Melindy Gatesów  
2000 – obecnie  
Współzałożyciel, prezes • Microsoft  
1975 – obecnie

### Wykształcenie

Uniwersytet Harvarda  
1973 – 1975  
Lakeside School, Seattle

### Dane kontaktowe

Blog: [thegatesnotes.com](http://thegatesnotes.com)  
Twitter: @BillGates

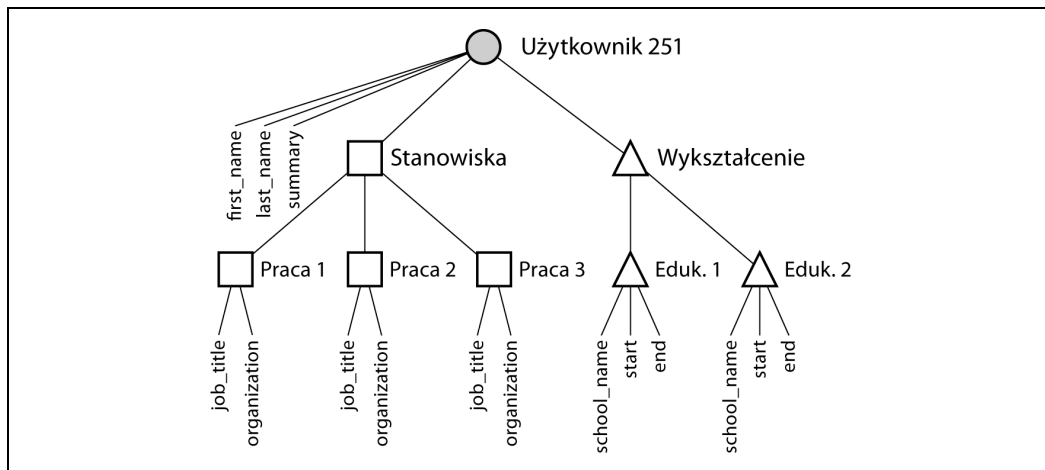


Rysunek 2.1. Reprezentowanie profilu z serwisu LinkedIn za pomocą schematu relacyjnego. Zdjęcie Billa Gatesa dzięki uprzejmości Wikimedia Commons (Ricardo Stuckert, Agência Brasil)

Niektórzy programiści uważają, że model JSON zmniejsza niezgodność impedancji między kodem aplikacji a warstwą składowania danych. Jednak jak się przekonasz z rozdziału 4., JSON jako format kodowania danych powoduje pewne problemy. Brak schematu jest często podawany jako zaleta. Zagadnienie to zostało opisane w punkcie „Elastyczność schematu w modelu opartym na dokumentach”.

Reprezentacja w formacie JSON jest bardziej *lokalna* niż wielotabelowy schemat z rysunku 2.1. Jeśli chcesz pobrać profil z bazy relacyjnej, musisz albo wykonać wiele zapytań (do każdej tabeli na podstawie pola `user_id`), albo przeprowadzić skomplikowane wielostronne złączenie między tabelą `users` a powiązаныmi tabelami. W reprezentacji w formacie JSON wszystkie potrzebne informacje znajdują się w jednym miejscu i wystarczy jedno zapytanie.

Relacje jeden do wielu między profilem użytkownika a stanowiskami użytkownika, historią wykształcenia i danymi kontaktowymi sugeruje drzewiastą strukturę danych, a reprezentacja w formacie JSON bezpośrednio ukazuje tę strukturę (rysunek 2.2).



Rysunek 2.2. Relacja jeden do wielu dająca strukturę drzewiastą

## Relacje wiele do jednego i wiele do wielu

Na listingu 2.1 z poprzedniego punktu w polach `region_id` i `industry_id` używane są identyfikatory zamiast zwykłego tekstu "Obszar metropolitalny Seattle" i "Filantropia". Dlaczego?

Jeśli interfejs użytkownika obejmuje pola tekstowe do podawania regionu i branży, sensowne jest zapisywanie tych danych jako zwykłego tekstu. Istnieją jednak zalety używania standardowych list regionów geograficznych i branż oraz umożliwiania użytkownikom wyboru z listy rozwijanej lub udostępnianie automatycznego uzupełniania. Oto te zalety:

- Spójny styl i pisownia w profilach.
- Unikanie wieloznaczności (np. jeśli istnieje kilka miast o tej samej nazwie).
- Łatwość aktualizowania. Nazwa jest zapisywana w tylko jednym miejscu, dlatego można ją łatwo zaktualizować w całym serwisie, jeśli konieczna będzie modyfikacja (np. z powodu zmiany nazwy miasta wskutek wydarzeń politycznych).
- Ułatwienie internacjonalizacji. Gdy witryna jest tłumaczona na inne języki, można przetłumaczyć standardowe listy i wyświetlać nazwy regionów oraz branż w języku użytkownika.
- Łatwiejsze wyszukiwanie. Na przykład wyszukiwanie filantropów ze stanu Waszyngton może zwrócić ten profil, ponieważ na liście regionów można zakodować to, że Seattle znajduje się w tym stanie (co nie jest oczywiste na podstawie tekstu "Obszar metropolitalny Seattle").

Używanie identyfikatorów lub zwykłego tekstu wiąże się z kwestią powielania danych. Gdy stosujesz identyfikator, informacje zrozumiałe dla ludzi (np. słowo *Filantropia*) są przechowywane tylko w jednym miejscu, a wszędzie, gdzie te informacje są potrzebne, podawany jest identyfikator (który ma znaczenie wyłącznie w ramach bazy). Jeśli bezpośrednio zapisujesz tekst, powielasz zrozumiałe dla człowieka informacje w każdym używającym ich rekordzie.

Zaleta stosowania identyfikatorów polega na tym, że ponieważ nie mają znaczenia dla ludzi, nigdy nie wymagają zmian. Identyfikator może pozostać taki sam nawet wtedy, gdy określane przez niego informacje się zmieniają. Wszystkie dane zrozumiałe dla ludzi mogą się w przyszłości zmienić.

Jeśli takie informacje są powielone, konieczna jest aktualizacja wszystkich ich kopii. To powoduje koszty związane z zapisem i ryzyko niespójności (jeśli niektóre kopie zostaną zaktualizowane, a inne nie). Eliminowanie takich duplikatów to główna idea *normalizacji* baz danych<sup>2</sup>.



Administratorzy baz danych i programiści uwielbiają dyskutować na temat normalizacji lub denormalizacji. Jednak na razie powstrzymam się od ocen. W części III wracam do tego zagadnienia i omawiam systematyczne metody radzenia sobie z pamięcią podręczną, denormalizacją i danymi pochodnymi.

Niestety, normalizacja przedstawionych danych wymaga relacji *wiele do jednego* (wiele osób mieszka w jednym regionie; wielu ludzi pracuje w jednej branży), co nie pasuje zbyt dobrze do modelu opartego na dokumentach. W bazach relacyjnych standardowo wskazuje się wiersze z innych tabel za pomocą identyfikatorów, ponieważ można łatwo tworzyć złączenia. W bazach opartych na dokumentach złączenia dla struktur drzewiastych typu jeden do wielu nie są potrzebne, a obsługa złączeń jest często słaba<sup>3</sup>.

Jeśli sama baza nie obsługuje złączeń, musisz zasymulować je w kodzie aplikacji, przesyłając wiele zapytań do bazy. W tym przykładzie lista regionów i branż jest prawdopodobnie krótka i zmienia się na tyle rzadko, że aplikacja może przechowywać ją w pamięci. Jednak i tak kod odpowiedzialny za złączenia jest przenoszony z bazy do aplikacji.

Ponadto nawet jeśli pierwotna wersja aplikacji zgadza się z wolnym od złączeń modelem opartym na dokumentach, zakres powiązań między danymi często rośnie wraz z dodawaniem do aplikacji nowych funkcji. Zastanów się nad zmianami, jakie mogą się pojawić w przykładzie dotyczącym życiorysu:

#### *Organizacje i szkoły jako encje*

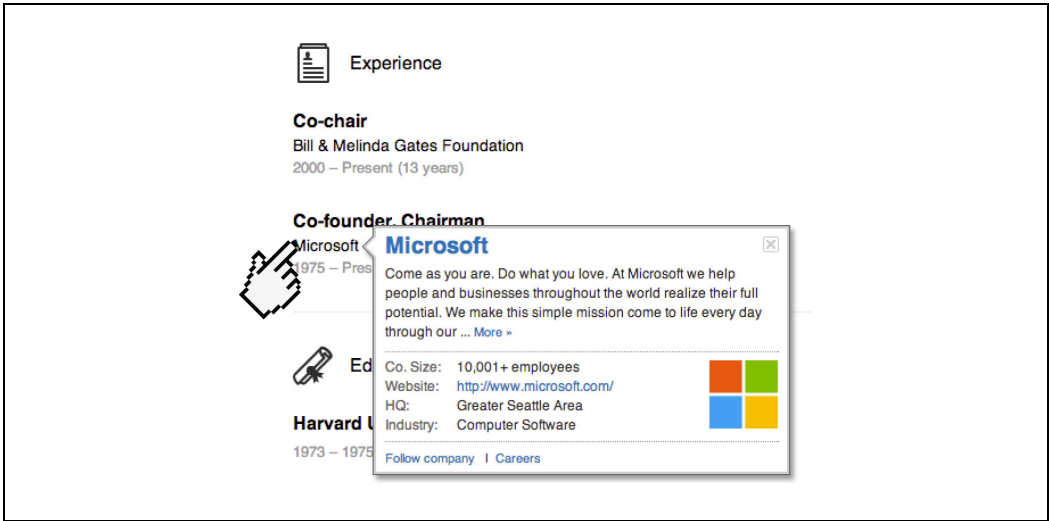
W poprzednim opisie pola `organization` (firma, w której użytkownik pracował) i `school_name` (miejsce, gdzie się uczył) zawierają zwykły tekst. A może umieścić tam referencje do encji? Wtedy dla każdej organizacji, szkoły lub uczelni można utworzyć odrębną stronę internetową (z logo, wiadomościami itd.). W każdym życiorysie można umieścić odsyłacz do wymienionych organizacji i szkół oraz dodać logo i inne informacje na temat tych jednostek (na rysunku 2.3 pokazano przykład z serwisu LinkedIn).

#### *Rekomendacje*

Załóżmy, że chcesz dodać nową funkcję — jeden użytkownik może zarekomendować innego. Rekomendacja jest wyświetlana w życiorysie polecanego użytkownika razem z nazwiskiem i zdjęciem autora rekomendacji. Jeśli autor rekomendacji zaktualizuje zdjęcie, we wszystkich napisanych przez niego opiniach trzeba odzwierciedlić nową fotografię. Dlatego w rekomendacji należy umieścić referencję do profilu jej autora.

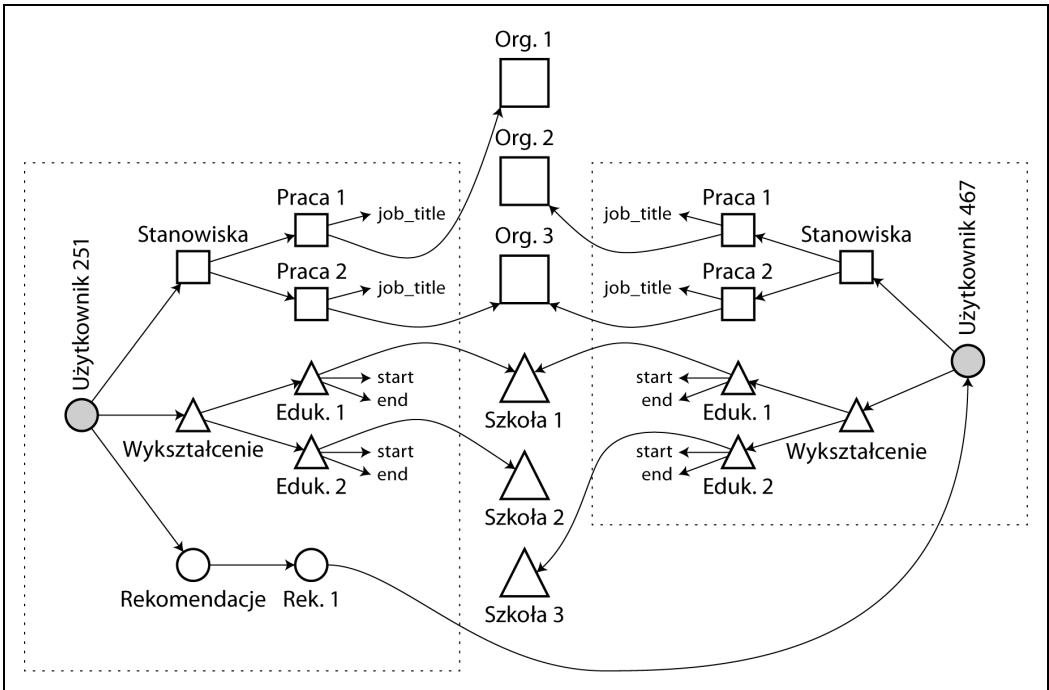
<sup>2</sup> W literaturze poświęconej modelowi relacyjnemu wyróżnia się kilka różnych postaci normalnych, jednak nie ma to dużego znaczenia praktycznego. Zgodnie z ogólną regułą jeśli powielasz wartości, które można zapisać w tylko jednym miejscu, schemat nie jest znormalizowany.

<sup>3</sup> W czasie powstawania tej książki złączenia są obsługiwane w bazie RethinkDB, nie są dostępne w bazie MongoDB, a w bazie CouchDB działają tylko dla statycznych widoków.



Rysunek 2.3. Nazwa firmy to nie zwykły łańcuch znaków, ale odsyłacz do encji reprezentującej firmę (zrzut z serwisu linkedin.com)

Na rysunku 2.4 pokazano, że nowe funkcje wymagają relacji wiele do wielu. Dane z każdego prostokąta z przerywaną ramką można pogrupować w jednym dokumencie, ale wystąpienia organizacji, szkół i innych użytkowników trzeba przedstawić w formie referencji, co wymaga złączeń w relacji na zapytanie.



Rysunek 2.4. Rozbudowywanie życiorysów o relacje wiele do wielu

## Czy bazy oparte na dokumentach to powtórka z historii?

Choć w bazach relacyjnych relacje wiele do wielu i złączenia są powszechnie używane, bazy oparte na dokumentach i model NoSQL spowodowały powrót do dyskusji na temat optymalnego reprezentowania takich relacji w bazach. Ta dyskusja jest znacznie starsza niż model NoSQL. Jej początki sięgają najstarszych skomputeryzowanych systemów bazodanowych.

W latach 70. najpopularniejszą bazą do przetwarzania danych biznesowych była **IMS** (ang. *Information Management System*) firmy IBM, pierwotnie opracowana na potrzeby zarządzania magazynami w programie kosmicznym Apollo, a po raz pierwszy udostępniona komercyjnie w 1968 r. [13]. Ta baza wciąż jest używana i konserwowana. Działa w systemie OS/390 na maszynach typu mainframe IBM-a [14].

W projekcie bazy IMS wykorzystano stosunkowo prosty model danych, *model hierarchiczny*, zaskakująco podobny do modelu wykorzystującego format JSON z baz opartych na dokumentach [2]. W tym projekcie wszystkie dane były reprezentowane jako drzewo rekordów zagnieżdżonych w innych rekordach (na podobieństwo struktury w formacie JSON z rysunku 2.2).

Baza IMS, podobnie jak bazy oparte na dokumentach, działała dobrze dla relacji jeden do wielu, jednak miała trudności z relacjami wiele do wielu i nie obsługiwała złączeń. Programiści musieli zdecydować, czy powielić (denormalizować) dane, czy ręcznie przetwarzać referencje prowadzące do innych rekordów. Te problemy z lat 60. i 70. są bardzo podobne do trudności, na jakie programiści natrafiają obecnie, pracując z bazami opartymi na dokumentach [15].

Zaproponowano różne rozwiązania problemu ograniczeń modelu hierarchicznego. Dwa najważniejsze z tych rozwiązań to *model relacyjny* (który został przekształcony w SQL i zdominował świat) i *model sieciowy* (który początkowo zyskał wielu zwolenników, ale ostatecznie odszedł w niepamięć). „Wielki spór” między tymi dwoma obozami trwał przez dużą część lat 70. [2].

Ponieważ problem, które te dwa modele miały rozwiązać, obecnie wciąż jest aktualny, warto po krótku wrócić do tego sporu, patrząc na niego z obecnej perspektywy.

### Model sieciowy

Model sieciowy został ustandaryzowany przez komitet **CODASYL** (ang. *Conference on Data Systems Languages*) i zaimplementowany przez kilku różnych producentów baz danych. Inna nazwa tego podejścia to *model CODASYL* [16].

Model CODASYL był uogólnieniem modelu hierarchicznego. W strukturze drzewiastej modelu hierarchicznego występuje dokładnie jeden element nadrzędny. W modelu sieciowym rekord może mieć wiele elementów nadrzędnych. Może np. występować rekord reprezentujący region "Obszar metropolitalny Seattle", z którym powiązany jest każdy użytkownik mieszkający na tym obszarze. Pozwala to tworzyć modele relacji wiele do jednego i wiele do wielu.

W modelu sieciowym powiązania między rekordami nie miały postaci kluczy obcych, raczej przypominały wskaźniki z języka programowania (choć były przechowywane na dysku). Jedyny sposób na dostęp do rekordu polegał na przejściu ścieżki z korzenia wzdłuż łańcucha powiązań. Była to tzw. *ścieżka dostępu*.

W najprostszej sytuacji przechodzenie ścieżką dostępu przypominało poruszanie się na liście powiązanej. Należy zacząć od głowy listy i sprawdzać kolejne rekordy do momentu znalezienia pożądanego. Jednak w świecie relacji wiele do wielu ten sam rekord można osiągnąć za pomocą kilku różnych ścieżek dostępu, a programista pracujący z modelem sieciowym musi o tym pamiętać.

Zapytanie w modelu CODASYL jest obsługiwane przez przenoszenie kursora po bazie. Wymaga to poruszania się po listach rekordów i podążania ścieżkami dostępu. Jeśli rekord ma kilka elementów nadrzędnych (czyli parę prowadzących do niego wskaźników z innych rekordów), kod aplikacji musi śledzić wszystkie te różne relacje. Nawet członkowie komitetu CODASYL przyznawali, że przypomina to poruszanie się w  $n$ -wymiarowej przestrzeni danych [17].

Choć ręczny wybór ścieżki dostępu pozwalał na najwydajniejsze wykorzystanie bardzo ograniczonych możliwości sprzętowych w latach 70. (np. napędów taśmowych, na których wyszukiwanie było bardzo wolne), problem polegał na tym, że kod do zgłaszania zapytań i aktualizowania bazy był skomplikowany i mało podatny na zmiany. W obu modelach (hierarchicznym i sieciowym) brak ścieżki do potrzebnych danych oznaczał trudności. Można było zmienić ścieżki dostępu, jednak wymagało to sprawdzenia dużej ilości ręcznie napisanego kodu zapytań i zmodyfikowania go pod kątem obsługi nowych ścieżek. Wprowadzanie zmian w modelu danych aplikacji było więc trudne.

## Model relacyjny

Z kolei w modelu relacyjnym dostępne były wszystkie dane. Relacja (tabela) to po prostu kolekcja krotek (wierszy). Nie występują tu przypominające labirynt zagnieżdżone struktury ani skomplikowane ścieżki dostępu, którymi trzeba się poruszać, aby znaleźć dane. Możesz wczytać dowolne lub wszystkie wiersze tabeli, wybierając te pasujące do arbitralnie określonego warunku. Możesz też wczytać konkretny wiersz, tworząc kolumny pełniące funkcję klucza i dopasowujące dane na podstawie tych kolumn. Ponadto możesz wstawić nowy wiersz do dowolnej tabeli, nie martwiąc się o oparte na kluczu obcych relacje z innymi tabelami<sup>4</sup>.

W bazie relacyjnej optymalizator zapytań automatycznie ustala, w jakiej kolejności wykonywać poszczególne części zapytania i których indeksów używać. Te wybory można uznać za „ścieżkę dostępu”, jednak istotna różnica polega na tym, że są one dokonywane automatycznie przez optymalizator zapytań, a nie przez programistę aplikacji, dlatego bardzo rzadko trzeba się nad nimi zastanawiać.

Jeśli chcesz tworzyć zapytania o dane w nowy sposób, możesz zadeklarować nowy indeks, a zapytania automatycznie wybiorą najbardziej odpowiedni z indeksów. Nie musisz modyfikować zapytań, aby wykorzystać nowy indeks (zob. też punkt „Języki zapytań o dane”). Dlatego model relacyjny znacznie ułatwia dodawanie do aplikacji nowych funkcji.

Optymalizatory zapytań w relacyjnych bazach danych to skomplikowane mechanizmy, a badania nad nimi i ich rozwój zajęły wiele lat [18]. Oto ważne spostrzeżenie związane z modelem relacyjnym: optymalizator zapytań wystarczy zbudować raz, a następnie mogą korzystać z niego wszystkie aplika-

---

<sup>4</sup> Ograniczenie w postaci klucza obcego pozwala ograniczać modyfikacje. Jednak w modelu relacyjnym takie ograniczenia nie są wymagane. Nawet gdy te ograniczenia występują, złączenia z użyciem kluczy obcych odbywa się w czasie przetwarzania zapytania, natomiast w modelu CODASYL złączenie odbywało się na etapie wstawiania danych.

cje używające określonej bazy. Jeśli nie masz optymalizatora zapytań, łatwiej ręcznie zapisać ścieżki dostępu w konkretnym zapytaniu niż opracować optymalizator do ogólnego użytku. Jednak w dłuższej perspektywie rozwiązanie o ogólnym przeznaczeniu wygrywa.

## Porównanie z bazami opartymi na dokumentach

Bazy oparte na dokumentach oznaczają cofnięcie do modelu hierarchicznego w jednym aspekcie: przechowywania rekordów zagnieżdżonych (relacji jeden do wielu takich jak pola `positions`, `education` i `contact_info` z rysunku 2.1) w rekordzie nadrzędnym zamiast w odrębnej tabeli.

Jednak w kontekście reprezentowania relacji wiele do jednego i wiele do wielu bazy relacyjne i oparte na dokumentach zasadniczo nie różnią się między sobą. W obu sytuacjach powiązany element jest wskazywany za pomocą unikatowego identyfikatora. W modelu relacyjnym nazywa się go *kluczem obcym*, a w modelu opartym na dokumentach — *referencją do dokumentu* [9]. Ten identyfikator jest określany w momencie odczytu za pomocą złączenia lub zapytań pomocniczych. Do tej pory bazy oparte na dokumentach nie podzieliły losu modelu CODASYL.

## Bazy relacyjne a bazy oparte na dokumentach obecnie

Między bazami relacyjnymi a bazami opartymi na dokumentach występuje wiele różnic, w tym aspekty odporności na błędy (zob. rozdział 5.) i obsługa równoległości (zob. rozdział 7.). W tym rozdziale analizowane są tylko różnice w modelu danych.

Głównymi argumentami na rzecz modelu danych opartego na dokumentach są elastyczność schematu, wyższa wydajność dzięki lokalnemu przechowywaniu danych i to, że w niektórych aplikacjach taki model jest bardziej podobny do struktur danych używanych przez aplikację. Model relacyjny zapewnia lepszą obsługę złączeń i relacje wiele do jednego oraz wiele do wielu.

### Który model danych prowadzi do prostszego kodu aplikacji?

Jeśli dane w aplikacji mają strukturę podobną do dokumentu (np. drzewa relacji jeden do wielu, gdzie zwykle całe drzewo wczytuje się od razu), prawdopodobnie dobry pomysł stanowi zastosowanie modelu opartego na dokumencie. Relacyjna technika *dzielenia danych* (ang. *shredding*), czyli rozbijania struktury podobnej do dokumentu na wiele tabel (takich jak `positions`, `education` i `contact_info` z rysunku 2.1), może prowadzić do kłopotliwych schematów i niepotrzebnie skomplikowanego kodu aplikacji.

Model oparty na dokumentach ma ograniczenia. Nie możesz np. bezpośrednio wskazać zagnieżdżonego elementu w dokumencie. Zamiast tego musisz zastosować zwrot taki jak „drugi element na liście stanowisk użytkownika nr 251” (co bardzo przypomina ścieżkę dostępu z modelu hierarchicznego). Jednak dopóki zagnieżdżenie w dokumencie nie jest zbyt głębokie, nie stanowi to problemu.

Słaba obsługa złączeń w bazach opartych na dokumentach może rodzić kłopoty, ale nie zawsze tak jest (zależy to od aplikacji). Na przykład relacje wiele do wielu mogą nie być potrzebne w aplikacji analitycznej, która używa opartej na dokumentach bazy do rejestrowania, jakie zdarzenia wystąpiły w określonym czasie [19].

Jeśli jednak aplikacja wykorzystuje relacje wiele do wielu, model oparty na dokumentach staje się mniej atrakcyjny. Można ograniczyć potrzebę złączeń za pomocą denormalizacji, jednak wtedy kod aplikacji będzie musiał wykonywać dodatkową pracę, aby zachować spójność zdenormalizowanych danych. Złączenia można symulować w kodzie aplikacji, zgłaszając wiele żądań do bazy. Jednak także to powoduje przeniesienie złożoności do aplikacji, a takie rozwiązanie jest zwykle wolniejsze niż przeprowadzanie złączeń przez wyspecjalizowany kod w bazie. W takich sytuacjach używanie modelu opartego na dokumentach może skutkować dużo bardziej skomplikowanym kodem aplikacji i niższą wydajnością [15].

Nie da się ogólnie stwierdzić, który model danych prowadzi do prostszego kodu aplikacji. Zależy to od rodzaju relacji między elementami danych. Gdy między danymi występuje wiele powiązań, model oparty na dokumentach jest kłopotliwy, model relacyjny jest akceptowalny, a modele oparte na grafach (zob. punkt „Modele danych zbliżone do grafu”) są najbardziej naturalne.

### Elastyczność schematu w modelu opartym na dokumentach

Większość baz opartych na dokumentach, a także format JSON w bazach relacyjnych, nie wymuszają stosowania określonego schematu do danych z dokumentów. Format XML w bazach relacyjnych zwykle jest powiązany z opcjonalnym sprawdzaniem poprawności schematu. Brak schematu oznacza, że do dokumentu można dodawać dowolne klucze i wartości, a w trakcie odczytu klienci nie mają gwarancji co do tego, jakie pola mogą się znajdować w dokumentach.

Bazy oparte na dokumentach są czasem nazywane *pozbawionymi schematu* (ang. *schemaless*), co bywa mylące, ponieważ kod wczytujący dane zwykle przyjmuje występowanie określonej struktury. Występuje więc niejawnie określany schemat, który jednak nie jest wymuszany przez bazę [20]. Precyzyjniejsze określenie stanowi zatem nazwa *schemat z etapu odczytu* (struktura danych jest domniemana i określana dopiero w momencie ich odczytu) pozostająca w kontraście do *schematu z etapu zapisu* (jest to tradycyjne podejście z baz relacyjnych, gdzie schemat podaje się bezpośrednio, a baza gwarantuje, że wszystkie zapisywane dane są z nim zgodne) [21].

Schemat z etapu odczytu przypomina dynamiczne sprawdzanie typów (w czasie wykonywania programu) w językach programowania, a schemat z etapu zapisu jest podobny do statycznego sprawdzania typów (z czasu kompilacji). Podobnie jak zwolennicy statycznego i dynamicznego sprawdzania typów prowadzą spory na temat względnych zalet tych podejść [22], tak i wymuszanie zgodności ze schematami w bazach to kontrowersyjny temat. Na ogólnym poziomie nie istnieją złe ani dobre rozwiązania.

Różnica między omawianymi podejściami jest zauważalna zwłaszcza w sytuacjach, gdy w aplikacji zmienia się format danych. Załóżmy, że obecnie przechowujesz imię i nazwisko każdego użytkownika w jednym polu, ale chcesz rozdzielić te elementy [23]. W bazie opartej na dokumentach wystarczy zacząć zapisywać nowe dokumenty z nowymi polami, a w kodzie aplikacji obsługiwać odczyt starszych dokumentów. Oto przykład:

```
if (user && user.name && !user.first_name) {  
    // Dokumenty zapisane przed 8 grudnia 2013 nie obejmują pola first_name  
    user.first_name = user.name.split(" ")[0];  
}
```



W schematach baz ze „statyczną kontrolą typów” zwykle przeprowadzana jest *migracja* w sposób podobny do poniższego:

```
ALTER TABLE users ADD COLUMN first_name text;  
UPDATE users SET first_name = split_part(name, ' ', 1); -- PostgreSQL  
UPDATE users SET first_name = substring_index(name, ' ', 1); -- MySQL
```

Modyfikowanie schematu ma złą opinię, ponieważ uważa się, że długo trwa i wymaga przestoju aplikacji. Ta ocena nie jest w pełni zasłużona. Większość baz relacyjnych wykonuje instrukcję ALTER TABLE w ciągu milisekund. Istotny wyjątek stanowi tu baza MySQL, która po wywołaniu instrukcji ALTER TABLE kopiuje całą tabelę, co przy modyfikowaniu dużej tabeli może oznaczać minuty, a nawet godziny przestoju. Istnieją jednak różne narzędzia pomagające radzić sobie z tym ograniczeniem [24, 25, 26].

Wykonywanie instrukcji UPDATE na dużej tabeli zwykle trwa długo w każdej bazie, ponieważ trzeba zmodyfikować każdy wiersz. Jeśli jest to nieakceptowalne, aplikacja może pozostawić w polu first\_name wartość domyślną NULL i uzupełniać ją w momencie odczytu — tak jak to robi baza oparta na dokumentach.

Schemat z etapu odczytu ma zalety, jeśli z jakiegoś powodu elementy kolekcji nie mają tej samej struktury (dane są niejednorodne). Może to wynikać z następujących przyczyn:

- Występuje wiele różnych typów obiektów i niepraktyczne jest umieszczanie każdego z nich w odrębnej tabeli.
- Struktura danych jest określana przez zewnętrzne systemy, nad którymi nie masz kontroli i które mogą się zmienić w dowolnym momencie.

W takich sytuacjach schemat może być bardziej szkodliwy niż pomocny, a pozbawione schematu dokumenty mogą stanowić dużo bardziej naturalny model danych. Jednak kiedy oczekuje się, że wszystkie rekordy będą miały tę samą strukturę, schematy są przydatnym mechanizmem dokumentowania i wymuszania takiej struktury. Szczegółowe omówienie schematów i ich modyfikowania zawiera rozdział 4.

## Lokalność danych w kontekście zapytań

Dokument jest zwykle przechowywany jako jeden ciągły łańcuch znaków w formacie JSON, XML lub w ich binarnej odmianie (takiej jak BSON w bazie MongoDB). Jeśli aplikacja często potrzebuje dostępu do całego dokumentu (np. w celu wyświetlenia go na stronie internetowej), taka *lokalność przechowywania* przynosi korzyści w zakresie wydajności. Jeżeli dane są rozdzielone między wiele tabel (jak na rysunku 2.1), pobranie wszystkich danych wymaga wielu operacji wyszukiwania z użyciem indeksu. Może to wymagać więcej operacji wyszukiwania na dysku i więcej czasu.

Korzyści płynące z lokalności występują tylko wtedy, gdy potrzebujesz dużych porcji dokumentu. Baza zwykle musi wczytać cały dokument nawet wtedy, jeśli potrzebujesz tylko jego niewielkiej części, co w przypadku dużych dokumentów może oznaczać marnotrawstwo. Aktualizacja dokumentu zwykle wymaga ponownego zapisania go w całości. Jedynie modyfikacje, które nie zmieniają wielkości zakodowanego dokumentu, można łatwo przeprowadzić w miejscu [19]. Dlatego zwykle zaleca się, aby tworzyć stosunkowo niewielkie dokumenty i unikać zwiększania ich wielkości [9]. Ograniczenia związane z wydajnością znacznie zmniejszają zbiór sytuacji, w których bazy oparte na dokumentach są przydatne.

Warto podkreślić, że idea grupowania powiązanych danych w celu zwiększenia ich lokalności występuje nie tylko w modelu opartym na dokumentach. Na przykład baza Spanner Google’a zapewnia lokalność w modelu relacyjnym, ponieważ umożliwia zadeklarowanie w schemacie, że większe tabele powinny być zagnieżdżone w tabeli nadrzędnej [27]. System Oracle też to umożliwia, udostępniając mechanizm *tabel klastrowych z indeksem wielotabelowym* [28]. Mechanizm *rodziny kolumn* w modelu Bigtable (używanym w bazach Cassandra i HBase) też jest przeznaczony do zarządzania lokalnością [29].

Więcej o lokalności dowiesz się z rozdziału 3.

## Konwergencja baz relacyjnych i opartych na dokumentach

Większość baz relacyjnych (oprócz MySQL-a) obsługuje format XML od ok. 2005 r. Dostępne są m.in. funkcje wprowadzające lokalne modyfikacje w dokumentach XML-owych. Możliwe jest też indeksowanie dokumentów XML-owych i kierowanie zapytań o nie. Dzięki temu aplikacje mogą używać modeli danych bardzo podobnych do tych stosowanych razem z bazami opartymi na dokumentach.

PostgreSQL od wersji 9.3 [8], MySQL od wersji 5.7 i IBM DB2 od wersji 10.5 [30] zapewniają podobną obsługę dokumentów w formacie JSON. Ponieważ JSON to popularny internetowy interfejs API, możliwe, że także w innych bazach relacyjnych pojawi się obsługa tego formatu.

Jeśli chodzi o bazy oparte na dokumentach, RethinkDB obsługuje w języku zapytań złączenia podobne do tych z baz relacyjnych. Ponadto niektóre sterowniki bazy MongoDB automatycznie określają referencje w bazach (w praktyce przeprowadzając w ten sposób złączenia po stronie klienta, choć zwykle wykonywane są one wolniej niż złączenia w bazach, ponieważ wymagają dodatkowej wymiany danych w sieci i są mniej zoptymalizowane).

Wydaje się, że bazy relacyjne i oparte na dokumentach z czasem upodabniają się do siebie. Jest to korzystne — modele danych uzupełniają się<sup>5</sup>. Jeśli baza potrafi obsługiwać dane przypominające dokumenty, a także wykonywać na nich zapytania relacyjne, aplikacje mogą wykorzystać zestaw funkcji najlepiej dostosowany do potrzeb.

Hybrydowe połączenie modeli relacyjnego i dokumentowego to dobry pomysł na rozwój baz w przyszłości.

## Język zapytań o dane

Gdy wprowadzono model relacyjny, obejmował on nowy sposób zgłaszania zapytań o dane. SQL jest *deklaratywnym* językiem zapytań, podczas gdy IMS i CODASYL używają kodu *imperatywnego*. Co to oznacza?

---

<sup>5</sup> Przedstawiony przez Codd’a pierwotny opis modelu relacyjnego [1] umożliwiał stosowanie w schemacie relacyjnym czegoś podobnego do dokumentów w formacie JSON. Codd nazwał to *dziedzinaми nieprostymi* (ang. *nonsimple domains*). Chodziło o to, że wartość w wierszu nie musi być prostego typu danych (nie musi być liczbą lub łańcuchem znaków), ale może też być zagnieżdżoną relacją (tabelą). Pozwala to zastosować jako wartość dowolnie zagnieżdżoną strukturę drzewiastą — podobnie jak dzięki dodaniu do SQL-a obsługi formatów JSON lub XML 30 lat później.

Wiele popularnych języków programowania to języki imperatywne. Na przykład jeśli dostępna jest lista gatunków zwierząt, możesz napisać następujący kod, aby pobrać same rekiny:

```
function getSharks() {  
  var sharks = [];  
  for (var i = 0; i < animals.length; i++) {  
    if (animals[i].family === "Rekiny") {  
      sharks.push(animals[i]);  
    }  
  }  
  return sharks;  
}
```

Te dane w algebrze relacji wyglądałyby tak:

```
sharks =  $\sigma_{\text{family} = \text{"Rekiny"}}$ (animals)
```

Tu  $\sigma$  (grecka litera sigma) to operator selekcji zwracający tylko te zwierzęta, które pasują do warunku `family = "Rekiny"`.

SQL zdefiniowano w taki sposób, że dobrze odzwierciedla strukturę algebry relacji:

```
SELECT * FROM animals WHERE family = 'Rekiny';
```

Język imperatywny nakazuje komputerowi wykonanie określonych operacji w ustalonej kolejności. Możesz wyobrazić sobie uruchamianie kodu wiersz po wierszu, sprawdzanie warunków, aktualizowanie zmiennych i decydowanie, czy jeszcze raz wykonać pętlę.

W deklaratywnym języku zapytań, takim jak SQL lub algebra relacji, podają się tylko wzorzec potrzebnych danych — jakie warunki muszą spełniać wyniki i jak dane mają zostać przekształcone (np. posortowane, pogrupowane lub zregrowane). Nie podaje się, *jak* osiągnąć ten cel. To optymalizator zapytań z systemu bazodanowego decyduje, które indeksy i jakie metody złączania zastosować oraz w jakiej kolejności wykonywać różne części zapytania.

Deklaratywny język zapytań jest atrakcyjny, ponieważ jest zwykle bardziej zwięzły i łatwiejszy w użyciu niż imperatywne interfejsy API. Bardziej liczy się jednak to, że ukrywa szczegóły implementacji systemu bazodanowego, co pozwala zwiększać wydajność tego systemu bez wymogu modyfikowania zapytań.

Na przykład w kodzie imperatywnym przedstawionym na początku tego punktu gatunki występują na liście w określonej kolejności. Jeśli baza chce na zapleczu odzyskać nieużywaną przestrzeń dyskową, konieczne może być przestawienie rekordów, co zmienia kolejność gatunków. Czy baza może bezpiecznie wykonać to zadanie bez naruszania pracy zapytań?

Przykład w SQL-u nie gwarantuje kolejności gatunków, dlatego jej zmiana nie ma znaczenia. Jeśli jednak zapytanie jest napisane w kodzie imperatywnym, w bazie nigdy nie ma pewności, czy kod wymaga określonego uporządkowania elementów. Ponieważ funkcje SQL-a są ograniczone, możliwości automatycznej optymalizacji w bazie są większe.

Język deklaratywny często umożliwia równoległe wykonywanie kodu. Obecnie procesory stają się szybsze dzięki dodawaniu rdzeni, a nie w wyniku znacznego zwiększania taktowania [31]. Kod imperatywny bardzo trudno jest przetwarzać równoległe z użyciem wielu rdzeni i maszyn, ponieważ określa instrukcje, które trzeba wykonywać w konkretnej kolejności. Języki deklaratywne z większym prawdopodobieństwem będą szybciej wykonywane równoległe, ponieważ określają

tylko wzorce wyników, a nie algorytm używany do ich uzyskania. Baza może się swobodnie posłużyć równoległą implementacją języka zapytań, jeśli jest to wskazane [32].

## Zapytania deklaratywne w internecie

Zalety deklaratywnych języków zapytań nie ograniczają się do baz danych. Aby to udowodnić, warto porównać podejścia deklaratywne i imperatywne w zupełnie innym środowisku — w przeglądarce internetowej.

Założmy, że tworzysz witrynę poświęconą zwierzętom oceanicznym. Użytkownik przegląda stronę o rekinach, dlatego oznaczasz w menu opcję „Rekiny” jako aktualnie wybraną:

```
<ul>
  <li class="selected"> ❶
    <p>Rekiny</p> ❷
    <ul>
      <li>Żarłacz biały</li>
      <li>Żarłacz tygrysi</li>
      <li>Młotowate</li>
    </ul>
  </li>
  <li>
    <p>Wieloryby</p>
    <ul>
      <li>Płetwal błękitny</li>
      <li>Długopłetwiec oceaniczny</li>
      <li>Płetwal zwyczajny</li>
    </ul>
  </li>
</ul>
```

❶ Wybrany element jest wyróżniony klasą CSS "selected".

❷ Tytuł obecnie otwartej strony to <p>Rekiny</p>.

Założmy, że chcesz, aby tytuł otwartej strony miał niebieskie tło i wyróżniał się wizualnie. Za pomocą stylów CSS można to łatwo osiągnąć:

```
li.selected > p {
  background-color: blue;
}
```

Tu selektor CSS `li.selected > p` deklaruje wzorzec elementów, dla których należy ustawić niebieskie tło. Są to wszystkie znaczniki `<p>`, których bezpośrednim elementem nadrzędnym jest znacznik `<li>` o klasie CSS `selected`. Element `<p>Rekiny</p>` z tego przykładu pasuje do tego wzorca, natomiast `<p>Wieloryby</p>` do niego nie pasuje, ponieważ jego element nadrzędny `<li>` nie ma atrybutu `class="selected"`.

Jeśli używasz XSL-a zamiast stylów CSS, możesz uzyskać podobny efekt:

```
<xsl:template match="li[@class='selected']/p">
  <fo:block background-color="blue">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Tu wyrażenie XPath `li[@class='selected']/p` jest analogiczne do selektora `li.selected > p` z poprzedniego przykładu. Cechę wspólną CSS-a i XSL-a stanowi fakt, że są to języki *deklaratywne* służące do określania stylu dokumentu.

Wyobraź sobie, jak wyglądałoby życie, gdybyś musiał zastosować podejście imperatywne. Jeśli używasz JavaScriptu i interfejsu API w postaci modelu DOM (ang. *Document Object Model*), efekt może wyglądać tak:

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
    if (liElements[i].className === "selected") {
        var children = liElements[i].childNodes;
        for (var j = 0; j < children.length; j++) {
            var child = children[j];
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
                child.setAttribute("style", "background-color: blue");
            }
        }
    }
}
```

Ten kod w JavaScriptcie imperatywnie ustawia niebieskie tło elementu `<p>Rekiny</p>`, to jednak okropne rozwiązanie. Nie tylko jest znacznie dłuższe i trudniejsze do zrozumienia niż odpowiedniki w językach CSS i XSL, ale ponadto występują w nim poważne problemy:

- Usunięcie klasy `selected` (np. w wyniku kliknięcia innej strony przez użytkownika) nie powoduje usunięcia koloru niebieskiego nawet po ponownym uruchomieniu kodu. Dlatego element pozostanie wyróżniony do czasu ponownego wczytania całej strony. W CSS-ie przeglądarka automatycznie wykrywa, gdy reguła `li.selected > p` nie jest już spełniona, i usuwa niebieskie tło bezpośrednio po usunięciu klasy `selected`.
- Jeśli chcesz wykorzystać nowy interfejs API, np. wywołanie `document.getElementsByClassName` ↪("selected") lub nawet `document.evaluate()` (co może poprawić wydajność), będziesz musiał napisać kod od nowa. Z kolei producenci przeglądarek mogą poprawić wydajność języka CSS lub XPath bez naruszania zgodności z kodem.

W przeglądarce używanie deklaratywnych stylów CSS jest znacznie korzystniejsze niż imperatywne operowanie stylami za pomocą JavaScriptu. Podobnie w bazach deklaratywne języki zapytań takie jak SQL okazują się znacznie lepsze od imperatywnych interfejsów API do obsługi zapytań<sup>6</sup>.

## Zapytania w modelu MapReduce

*MapReduce* to model programowania używany do masowego przetwarzania dużych ilości danych za pomocą wielu maszyn. Model ten został spopularyzowany przez firmę Google [33]. MapReduce w ograniczonej postaci jest obsługiwany przez niektóre magazyny danych typu NoSQL (w tym przez bazy MongoDB i CouchDB) jako mechanizm do przetwarzania zapytań z samym odczytem dotyczących wielu dokumentów.

Model MapReduce opisano szczegółowo w rozdziale 10. Tu pokrótce przedstawiono zastosowanie tego modelu w bazie MongoDB.

MapReduce nie jest ani deklaratywnym językiem zapytań, ani kompletnym imperatywnym interfejsem API do zgłaszania zapytań. To coś pośredniego: logika zapytań jest tu przedstawiana za

---

<sup>6</sup> IMS i CODASYL wykorzystują imperatywne interfejsy API do zgłaszania zapytań. W aplikacjach zwykle wykorzystywano kod w języku COBOL do przetwarzania rekordów z bazy jeden po drugim [2, 16].

pomocą fragmentów kodu, które są wielokrotnie wywoływane przez platformę wykonawczą. Model ten opiera się na funkcjach map (inna nazwa to collect) i reduce (inne nazwy to fold i inject) występujących w wielu funkcyjnych językach programowania.

W ramach przykładu wyobraź sobie, że jesteś oceanografem i dodajesz do bazy rekord z obserwacją za każdym razem, gdy zobaczysz zwierzę w oceanie. Teraz chcesz wygenerować raport informujący, ile rekinów zobaczyłeś w ciągu miesiąca.

W bazie PostgreSQL możesz zapisać takie zapytanie w następujący sposób:

```
SELECT date_trunc('month', observation_timestamp) AS observation_month, ❶  
       sum(num_animals) AS total_animals  
FROM observations  
WHERE family = 'Rekiny'  
GROUP BY observation_month;
```

❶ Funkcja `date_trunc('month', timestamp)` określa miesiąc kalendarzowy obejmujący datę timestamp i zwraca znacznik czasu reprezentujący początek tego miesiąca. Zaokrągla więc czas w dół do najbliższego miesiąca.

To zapytanie najpierw filtruje obserwacje, aby uwzględnić tylko gatunki z rodziny Rekiny, a następnie grupuje obserwacje według miesiąca kalendarzowego, w jakim wystąpiły. W ostatnim kroku zapytanie sumuje zwierzęta z wszystkich obserwacji z tego miesiąca.

Za pomocą mechanizmu MapReduce z bazy MongoDB to samo rozwiązanie można zapisać w następujący sposób:

```
db.observations.mapReduce(  
  function map() { ❷  
    var year = this.observationTimestamp.getFullYear();  
    var month = this.observationTimestamp.getMonth() + 1;  
    emit(year + "-" + month, this.numAnimals); ❸  
  },  
  function reduce(key, values) { ❹  
    return Array.sum(values); ❺  
  },  
  {  
    query: { family: "Rekiny" }, ❶  
    out: "monthlySharkReport" ❻  
  }  
);
```

❶ Filtr uwzględniający tylko gatunki rekinów można podać deklaratywnie (to specyficzne dla bazy MongoDB rozszerzenie modelu MapReduce).

❷ Funkcja `map` języka JavaScript jest wywoływana raz dla każdego dokumentu pasującego do pola `query`. Do `this` przypisuje się dany obiekt dokumentu.

❸ Funkcja `map` generuje klucz (łańcuch znaków obejmujący rok i miesiąc; np. "2013-12" lub "2014-1") i wartość (liczbę zwierząt w danej obserwacji).

❹ Pary klucz-wartość wygenerowane przez funkcję `map` są grupowane według klucza. Dla każdej pary klucz-wartość z tym samym kluczem (czyli z tym samym miesiącem i rokiem) funkcja `reduce` jest wywoływana jednokrotnie.

❺ Funkcja `reduce` dodaje liczby zwierząt z wszystkich obserwacji z danego miesiąca.

⑥ Ostateczne dane wyjściowe są zapisywane w kolekcji `monthlySharkReport`.

Załóżmy, że kolekcja `observations` obejmuje te dwa dokumenty:

```
{
  observationTimestamp: Date.parse("Mon, 25 Dec 1995 12:34:56 GMT"),
  family:      "Rekiny",
  species:     "Żarłacz biały",
  numAnimals: 3
}
{
  observationTimestamp: Date.parse("Tue, 12 Dec 1995 16:17:18 GMT"),
  family:      "Rekiny",
  species:     "Tawrosz piskowy",
  numAnimals: 4
}
```

Funkcja `map` jest wywoływana raz dla każdego dokumentu, co skutkuje wywołaniami `emit("1995-12", 3)` i `emit("1995-12", 4)`. Dalej następuje wywołanie `reduce("1995-12", [3, 4])`, co daje wynik 7.

Możliwości funkcji `map` i `reduce` są ograniczone. Muszą to być *czyste* funkcje — oznacza to, że mogą używać tylko przekazanych do nich danych wejściowych, nie mogą kierować dodatkowych zapytań do bazy i nie mogą powodować efektów ubocznych. Te ograniczenia umożliwiają bazie wykonywanie takich funkcji w dowolnym miejscu i w dowolnej kolejności oraz ponowne uruchamianie ich po awarii. Mimo to funkcje te są bardzo przydatne: mogą parsować łańcuchy znaków, wywoływać funkcje biblioteczne, wykonywać obliczenia itd.

MapReduce to stosunkowo niskopoziomowy model programowania służący do wykonywania kodu w rozproszonym środowisku klastra maszyn. Języki zapytań wyższego poziomu, np. SQL, można zaimplementować za pomocą potoku operacji z modelu MapReduce (zob. rozdział 10.), jednak istnieje też wiele rozproszonych implementacji SQL-a, które nie używają tego modelu. Zauważ, że w SQL-u nie ma nic, co zmusza do używania go na jednej maszynie, MapReduce to nie jedyne narzędzie do wykonywania zapytań w rozproszonym środowisku.

Możliwość zastosowania kodu w JavaScriptcie wewnątrz zapytania to bardzo cenna funkcja zaawansowanych zapytań, przy czym jest ona dostępna nie tylko w modelu MapReduce. Także niektóre bazy SQL-owe można rozszerzyć o funkcje JavaScriptu [34].

Problem z użytkowaniem modelu MapReduce polega na tym, że trzeba napisać dwie starannie skoordynowane funkcje w JavaScriptcie, co często może być trudniejsze niż napisanie jednego zapytania. Ponadto deklaratywny język zapytań daje optymalizatorowi zapytań więcej możliwości poprawy wydajności zapytania. Z tych powodów w bazie MongoDB 2.2 dodano obsługę deklaratywnego języka zapytań nazywanego *potokiem agregacji* [9]. W tym języku to samo zapytanie zliczające rekiny wygląda tak:

```
db.observations.aggregate([
  { $match: { family: "Rekiny" } },
  { $group: {
    _id: {
      year: { $year: "$observationTimestamp" },
      month: { $month: "$observationTimestamp" }
    },
    totalAnimals: { $sum: "$numAnimals" }
  } }
]);
```

Język potoku agregacji ma podobne możliwości jak podzbiór SQL-a, ale używa składni opartej na JSON-ie zamiast przypominającej zdania z języka angielskiego składni SQL-a. Wybór jednego z tych rozwiązań to głównie kwestia gustu. Morał z tej historii jest taki, że twórcy systemów NoSQL mogą przypadkowo odkryć SQL od nowa, pod inną nazwą.

## Modele danych przypominające graf

Wcześniej opisano, że relacje wiele do wielu są ważną cechą różniącą poszczególne modele danych. Jeśli w aplikacji występują głównie relacje jeden do wielu (dane o strukturze drzewiastej) lub relacje między rekordami w ogóle nie występują, model oparty na dokumentach jest odpowiedni.

Co jednak zrobić, jeśli w danych powszechne są relacje wiele do wielu? Model relacyjny radzi sobie z prostymi relacjami tego rodzaju, jednak gdy powiązania w danych są bardziej skomplikowane, naturalne staje się modelowanie danych za pomocą grafów.

Graf obejmuje dwa rodzaje obiektów: *wierzchołki* (nazywane też *węzłami* lub *encjami*) i *krawędzie* (inaczej *relacje* lub *łuki*). Za pomocą grafu można zamodelować wiele rodzajów danych. Oto typowe przykłady:

### *Grafy społeczne*

Wierzchołki reprezentują ludzi, a krawędzie określają, które osoby znają się ze sobą.

### *Graf stron internetowych*

Wierzchołki to strony internetowe, a krawędzie odpowiadają odsyłaczom HTML-owym do innych stron.

### *Sieci dróg lub torów kolejowych*

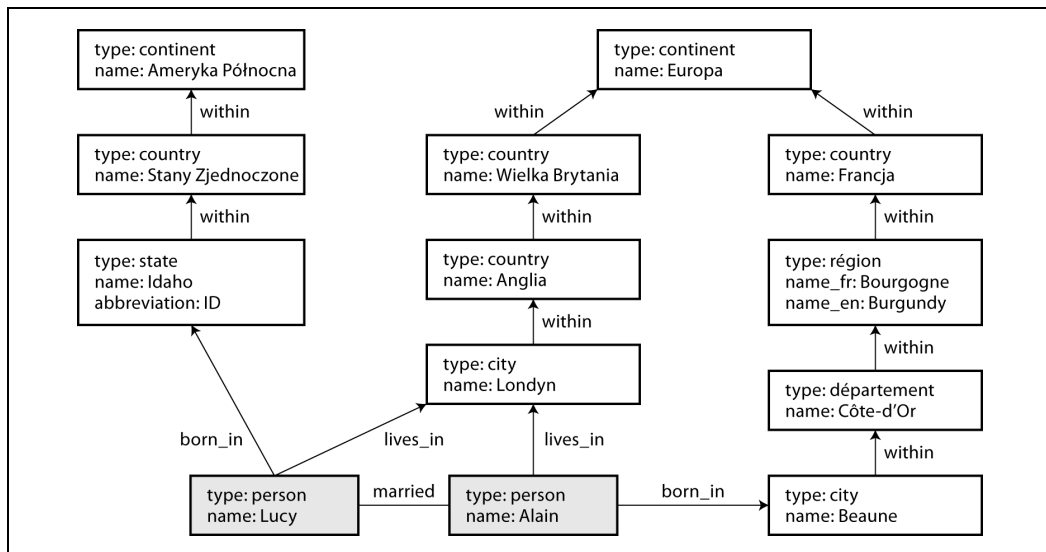
Wierzchołki to skrzyżowania, a krawędzie to łączące je drogi lub tory.

Operować na takich grafach mogą dobrze znane algorytmy. Na przykład samochodowe systemy nawigacji wyszukują najkrótszą ścieżkę między dwoma punktami w sieci dróg, a do grafu stron internetowych można zastosować algorytm PageRank, aby ustalić popularność strony, a tym samym i jej miejsce w wynikach wyszukiwania.

W wymienionych przykładach wszystkie wierzchołki grafu reprezentują obiekty tego samego rodzaju (ludzi, strony internetowe lub skrzyżowania dróg). Jednak grafy nie są ograniczone do danych *jednorodnych*. Równie wartościowym zastosowaniem grafów jest zapewnianie spójnego sposobu przechowywania obiektów zupełnie odmiennego rodzaju w jednym magazynie danych. Na przykład Facebook utrzymuje jeden graf z wieloma rodzajami wierzchołków i krawędzi. Wierzchołki reprezentują ludzi, miejsca, zdarzenia, operacje zameldowania się i komentarze użytkowników, a krawędzie informują o znajomościach między osobami, zameldowaniach w poszczególnych miejscach, kto skomentował poszczególne wpisy, kto wziął udział w określonym zdarzeniu itd. [35].

W tym punkcie omówiono przykład z rysunku 2.5. Te dane mogą pochodzić z sieci społecznościowej lub bazy danych genealogicznych. Widoczne są tu dwie osoby: Lucy z Idaho i Alain z Beaune we Francji — to małżeństwo mieszkające w Londynie.





Rysunek 2.5. Przykładowe dane o strukturze grafu (prostokąty reprezentują wierzchołki, a strzałki to krawędzie)

Istnieje kilka różnych, ale powiązanych sposobów tworzenia struktury danych w grafach i zgłaszania zapytań. W tym punkcie omawiane są modele *grafu właściwości* (używanego w systemach Neo4j, Titan i InfiniteGraph) i *triplestore* (stosowanego w systemach Datomic, AllegroGraph i in.). Poznasz tu trzy deklaratywne języki zapytań przeznaczone dla grafów: Cypher, SPARQL i Datalog. Obok nich istnieją też imperatywne języki zapytań używane do grafów (np. Gremlin [36]) i platformy przetwarzania grafów (np. Pregel). Ich opis zawiera rozdział 10.

## Grafy właściwości

W modelu grafu właściwości każdy wierzchołek obejmuje:

- unikatowy identyfikator;
- zbiór krawędzi wychodzących;
- zbiór krawędzi wchodzących;
- kolekcję właściwości (par klucz-wartość).

Każda krawędź obejmuje:

- unikatowy identyfikator;
- wierzchołek, od którego zaczyna się krawędź (*wierzchołek początkowy*; ang. *tail vertex*);
- wierzchołek, w którym kończy się krawędź (*wierzchołek docelowy*; ang. *head vertex*);
- etykietę opisującą rodzaj relacji między tymi dwoma wierzchołkami;
- kolekcję właściwości (par klucz-wartość).

Możesz sobie wyobrazić, że magazyn danych w postaci grafu składa się z dwóch tabel relacyjnych: jednej przeznaczonej na wierzchołki i drugiej na krawędzie, co przedstawia listing 2.2 (w tym schemacie do przechowywania właściwości wierzchołków i krawędzi używany jest typ danych

json z bazy PostgreSQL). Każdej krawędzi zapisywane są wierzchołki docelowy i początkowy. Jeśli chcesz uzyskać krawędzie wchodzące lub wychodzące wierzchołka, możesz zgłosić zapytanie o kolumny `head_vertex` i `tail_vertex` z tabeli `edges`.

Listing 2.2. Reprezentowanie grafu właściwości za pomocą schematu relacyjnego

```
CREATE TABLE vertices (  
    vertex_id integer PRIMARY KEY,  
    properties json  
);  
  
CREATE TABLE edges (  
    edge_id integer PRIMARY KEY,  
    tail_vertex integer REFERENCES vertices (vertex_id),  
    head_vertex integer REFERENCES vertices (vertex_id),  
    label text,  
    properties json  
);  
  
CREATE INDEX edges_tails ON edges (tail_vertex);  
CREATE INDEX edges_heads ON edges (head_vertex);
```

Oto niektóre ważne aspekty tego modelu:

1. Każdy wierzchołek może mieć krawędź łączącą go z dowolnym innym wierzchołkiem. Nie istnieje schemat ograniczający, jakiego rodzaju elementy mogą lub nie mogą być ze sobą powiązane.
2. Dla każdego wierzchołka można w wydajny sposób znaleźć krawędzie wchodzące i wychodzące, aby *poruszać się* po grafie, czyli podążać do przodu i wstecz ścieżką przechodzącą przez łańcuch wierzchołków. To dlatego na listingu 2.2 tworzone są indeksy dla obu kolumn: `tail_vertex` i `head_vertex`.
3. Używając różnych etykiet dla relacji rozmaitego rodzaju, można zapisać w jednym grafie kilka rodzajów informacji, a przy tym utrzymać przejrzysty model danych.

Te cechy sprawiają, że grafy oferują dużą swobodę w zakresie modelowania danych, co widać na rysunku 2.5. Na rysunku pokazano kilka rzeczy, które trudno byłoby przedstawić w tradycyjnym schemacie relacyjnym. Są to np. różnego rodzaju podziały administracyjne w poszczególnych krajach (we Francji występują *departamenty* i *regiony*, natomiast w Stanach Zjednoczonych *hrabstwa* i *stany*), dziwne wydarzenia w historii takie jak państwo w państwie (pomijam na razie zawiłości związane z suwerennością państw i narodów) oraz różny poziom szczegółowości danych (miejsce zamieszkania Lucy jest podane jako miasto, natomiast miejsce urodzenia tylko jako stan).

Możesz sobie wyobrazić, że graf zostanie rozbudowany o wiele innych faktów na temat Lucy i Alaina lub innych osób. Na przykład możesz podać w grafie alergię na pokarmy (dodając wierzchołki z alergenami i reprezentującą alergię krawędź między osobą a alergenem) oraz powiązać alergeny ze zbiorem wierzchołków określającym, które potrawy zawierają poszczególne substancje. Następnie można napisać zapytanie, aby ustalić, które potrawy są bezpieczne dla każdej osoby. Grafy ułatwiają wprowadzanie zmian. Gdy dodajesz funkcje do aplikacji, graf można łatwo rozbudować, aby uwzględnić modyfikacje w strukturach danych aplikacji.

## Język zapytań Cypher

Cypher to deklaracyjny język zapytań dla grafów właściwości, utworzony dla opartej na grafach bazy Neo4j [37]. Nazwa języka pochodzi od postaci z filmu *Matrix* i nie jest powiązana z szyframi z kryptografii [38].

Na listingu 2.3 pokazano zapytanie w języku Cypher wstawiające lewą część rysunku 2.5 do bazy opartej na grafach. Resztę grafu można dodać w podobny sposób (dla czytelności ten kod został pominięty). Każdy wierzchołek ma symboliczną nazwę, np. USA lub Idaho, a w pozostałych miejscach zapytania można za pomocą tych nazw tworzyć krawędzie między wierzchołkami, używając notacji ze strzałką. Instrukcja (Idaho) -[:WITHIN]-> (USA) tworzy krawędź z etykietą WITHIN, gdzie Idaho to węzeł początkowy, a USA to węzeł docelowy.

Listing 2.3. Podzbiór danych z rysunku 2.5 przedstawiony w zapytaniu w języku Cypher

```
CREATE
  (NAmerica:Location {name:'Ameryka Północna', type:'continent'}),
  (USA:Location {name:'Stany Zjednoczone', type:'country'}),
  (Idaho:Location {name:'Idaho', type:'state'}),
  (Lucy:Person {name:'Lucy'}),
  (Idaho) -[:WITHIN]-> (USA) -[:WITHIN]-> (NAmerica),
  (Lucy) -[:BORN_IN]-> (Idaho)
```

Po dodaniu do bazy wszystkich wierzchołków i krawędzi z rysunku 2.5 można zacząć zadawać ciekawe pytania — np. *znajdź imiona wszystkich osób, które wyemigrowały ze Stanów Zjednoczonych do Europy*. Oto bardziej precyzyjny opis: celem jest znalezienie wszystkich wierzchołków z krawędzią BORN\_IN prowadzącą do miejsca w Stanach Zjednoczonych i krawędzią LIVING\_IN prowadzącą do lokalizacji w Europie oraz zwrócenie właściwości name tych wierzchołków.

Na listingu 2.4 pokazano, jak przedstawić to zapytanie w języku Cypher. Tej samej notacji ze strzałką używa się w klauzuli MATCH do wyszukiwania wzorców w grafie. Instrukcja (person) -[:BORN\_IN]-> () dopasowuje dwa wierzchołki powiązane krawędzią z etykietą BORN\_IN. Wierzchołek początkowy tej krawędzi jest wiązany ze zmienną person, a wierzchołek docelowy pozostaje bez nazwy.

Listing 2.4. Zapytanie w języku Cypher znajdujące osoby, które wyemigrowały ze Stanów Zjednoczonych do Europy

```
MATCH
  (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:Location {name:'Stany Zjednoczone'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europa'})
RETURN person.name
```

To zapytanie można odczytać w poniżej przedstawiony sposób.

Znajdź każdy wierzchołek (nazywany person), który spełnia *oba* poniższe warunki:

1. Wierzchołek person ma wychodzącą krawędź BORN\_IN do innego wierzchołka. Z tego docelowego wierzchołka można za pomocą łańcucha wychodzących krawędzi WITHIN dotrzeć do wierzchołka typu Location, którego właściwość name ma wartość "Stany Zjednoczone".
2. Ten sam wierzchołek person ma krawędź wychodzącą LIVES\_IN. Wybierając tę krawędź, a następnie przechodząc łańcuchem wychodzących krawędzi WITHIN, ostatecznie można dotrzeć do wierzchołka typu Location, którego właściwość name ma wartość "Europa".

Dla każdego tego rodzaju wierzchołka person należy zwrócić właściwość name.

Istnieje kilka sposobów na wykonanie tego zapytania. Z podanego tu opisu wynika, że należy zacząć od sprawdzenia wszystkich osób z bazy i zbadania miejsca urodzenia oraz zamieszkania każdej z nich. Zwrócone mają zostać tylko osoby spełniające podane kryteria.

Jednak można też zaczynać od dwóch wierzchołków `Location` i poruszać się wstecz. Jeśli istnieje indeks dla właściwości `name`, zapewne można wydajnie znaleźć dwa wierzchołki reprezentujące Stany Zjednoczone i Europę. Następnie da się znaleźć wszystkie lokalizacje (stany, regiony, miasta itd.) w Stanach Zjednoczonych i Europie, wybierając wszystkie wchodzące krawędzie `WITHIN`. W ostatnim kroku należy znaleźć osoby. Można to zrobić za pomocą wchodzącej krawędzi `BORN_IN` lub `LIVES_IN` w jednym w wierzchołków reprezentujących lokalizację.

W sposób typowy dla deklaratywnego języka zapytań nie trzeba podawać takich szczegółów w trakcie pisania zapytania. To optymalizator zapytań automatycznie wybiera strategię, która zapewne okaże się najwydajniejsza. Dzięki temu można się zająć pisaniem reszty aplikacji.

## Zapytania o grafy w SQL-u

Na listingu 2.2 zasugerowano, że dane z grafu można przedstawić w bazie relacyjnej. Jednak czy po umieszczeniu takich danych w strukturze relacyjnej da się zgłaszać zapytania za pomocą SQL-a?

Odpowiedź brzmi: „Tak, ale z pewnymi trudnościami”. W bazie relacyjnej zwykle z góry wiadomo, jakie złączenia będą potrzebne w zapytaniu. W zapytaniach dotyczących grafów czasem trzeba pokonać nieznaną liczbę krawędzi przed znalezieniem szukanego wierzchołka. Oznacza to, że liczba złączeń nie jest z góry ustalona.

W omawianym przykładzie dzieje się tak w regule `() -[:WITHIN*0..]-> ()` w zapytaniu w języku Cypher. Krawędź `LIVES_IN` danej osoby może prowadzić do lokalizacji dowolnego rodzaju: ulicy, miasta, dystryktu, regionu, stanu itd. Miasto może się znajdować w (operator `WITHIN`) regionie, region w stanie, stan w państwie itd. Krawędź `LIVES_IN` może bezpośrednio prowadzić do szukanego wierzchołka z lokalizacją, ale może też być oddalona o kilka poziomów w hierarchii lokalizacji.

W języku Cypher kod `:WITHIN*0..` pozwala bardzo zwięźle to wyrazić. Ten kod oznacza: „przejdź krawędzią `WITHIN` zero lub więcej razy”. Taki zapis działa jak operator `*` w wyrażeniach regularnych.

Od wersji SQL:1999 ścieżki o zmiennej długości można zapisać w zapytaniach za pomocą *rekurencyjnego wspólnego wyrażenia tabelowego* (ang. *recursive common table expression*; składnia `WITH RECURSIVE`). Na listingu 2.5 to samo zapytanie (wyszukujące imiona osób, które wyemigrowały ze Stanów Zjednoczonych do Europy) jest zapisane w SQL-u za pomocą wspomnianej techniki (obsługiwanej w bazach PostgreSQL, IBM DB2, Oracle i SQL Server). Jednak ta składnia w porównaniu z rozwiązywaniem z języka Cypher jest bardzo nieporęczna.

*Listing 2.5. To samo zapytanie co z listingu 2.4 zapisane w SQL-u za pomocą rekurencyjnego wspólnego wyrażenia tabelowego*

```
WITH RECURSIVE
```

```
-- in_usa to zbiór identyfikatorów wierzchołków z wszystkimi lokalizacjami ze Stanów Zjednoczonych
in_usa(vertex_id) AS (
  SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Stany Zjednoczone' ❶
  UNION
```

```

SELECT edges.tail_vertex FROM edges ❷
JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
WHERE edges.label = 'within'
),

-- in_europe to zbiór identyfikatorów wierzchołków z wszystkimi lokalizacjami ze Stanów Zjednoczonych
in_europe(vertex_id) AS (
  SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europa' ❸
  UNION
  SELECT edges.tail_vertex FROM edges
  JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
  WHERE edges.label = 'within'
),

-- born_in_usa to zbiór identyfikatorów wierzchołków reprezentujących
-- wszystkie osoby urodzone w USA
born_in_usa(vertex_id) AS ( ❹
  SELECT edges.tail_vertex FROM edges
  JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
  WHERE edges.label = 'born_in'
),

-- lives_in_europe to zbiór identyfikatorów wierzchołków reprezentujących wszystkie
-- osoby żyjące w Europie
lives_in_europe(vertex_id) AS ( ❺
  SELECT edges.tail_vertex FROM edges
  JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
  WHERE edges.label = 'lives_in'
)

SELECT vertices.properties->>'name'
FROM vertices
-- Złączenie w celu znalezienia osób, które zarówno urodziły się w USA, *jak i* mieszkają w Europie
JOIN born_in_usa ON vertices.vertex_id = born_in_usa.vertex_id ❻
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;

```

❶ Najpierw znajdowany jest wierzchołek, którego właściwość name ma wartość "Stany Zjednoczone". Jest on zapisywany jako pierwszy element zbioru wierzchołków in\_usa.

❷ Kod podąża wszystkimi wchodzącymi krawędziami within z wierzchołków ze zbioru in\_usa i dodaje je do tego samego zbioru. Robi to do czasu odwiedzenia wszystkich krawędzi within.

❸ To samo należy zrobić, zaczynając od wierzchołka z właściwością name równą "Europa". W ten sposób budowany jest zbiór wierzchołków in\_europe.

❹ Dla każdego wierzchołka ze zbioru in\_usa należy przejść wchodzącymi krawędziami born\_in, aby znaleźć osobę urodzoną gdzieś w Stanach Zjednoczonych.

❺ Podobnie dla każdego wierzchołka ze zbioru in\_europe należy przejść wchodzącymi krawędziami lives\_in w celu wyszukiwania ludzi mieszkających w Europie.

❻ W ostatnim kroku za pomocą złączenia ustalana jest część wspólna zbiorów osób urodzonych w USA i mieszkających w Europie.

Jeśli to samo zapytanie można napisać za pomocą czterech wierszy w jednym języku i 29 wierszy w innym, pokazuje to, że różne modele danych są zaprojektowane pod kątem odmiennych przypadków użycia. Ważne, aby wybrać model danych dostosowany do aplikacji.

## Model triplestore i język SPARQL

Model triplestore jest w większości odpowiednikiem modelu grafu właściwości, przy czym do opisu tych samych pomysłów używane są inne określenia. Mimo to warto opisać ten model, ponieważ powiązane są z nim różne narzędzia i języki, które mogą się okazać przydatnymi dodatkami do Twojego zestawu rozwiązań do budowania aplikacji.

W modelu triplestore wszystkie informacje są przechowywane w formie bardzo prostych trzyczęściowych instrukcji (*podmiot, predykat, obiekt*). Na przykład w trójce (*Jan, lubi, banany*) *Jan* to podmiot, *lubi* to predykat (czasownik), a *banany* to obiekt.

Podmiot w trójce jest odpowiednikiem wierzchołka w grafie. Obiekt to jedna z dwóch rzeczy:

1. Wartość prostego typu danych, np. łańcuchem znaków lub liczbą. Wtedy predykat i obiekt z trójki są odpowiednikami klucza i wartości z właściwości wierzchołka odpowiadającego podmiotowi. Na przykład zapis (*maria, wiek, 33*) to odpowiednik wierzchołka *maria* o właściwości {"wiek":33}.
2. Inny wierzchołek grafu. Wtedy predykat to krawędź grafu, podmiot to wierzchołek początkowy, a obiekt to wierzchołek docelowy. Na przykład w trójce (*maria, zamężna, alan*) podmiot *maria* i obiekt *alan* to wierzchołki, a predykat *zamężna* to etykieta łączącej ich krawędzi.

Na listingu 2.6 pokazane są te same dane co na listingu 2.3, zapisane jako trójki w formacie *Turtle* (to podzbiór notacji *Notation3* — *N3* [39]).

Listing 2.6. Podzbiór danych z rysunku 2.5 przedstawiony jako trójki w formacie *Turtle*

```
@prefix : <urn:example:>.
_:lucy a :Person.
_:lucy :name "Lucy".
_:lucy :bornIn _:idaho.
_:idaho a :Location.
_:idaho :name "Idaho".
_:idaho :type "state".
_:idaho :within _:usa.
_:usa a :Location.
_:usa :name "Stany Zjednoczone".
_:usa :type "country".
_:usa :within _:namerica.
_:namerica a :Location.
_:namerica :name "Ameryka Północna".
_:namerica :type "continent".
```

W tym przykładzie wierzchołki grafu są zapisywane w formacie `_:nazwa`. Poza plikiem ta nazwa nie ma znaczenia. Istnieje tylko dlatego, że w przeciwnym razie nie byłoby wiadomo, które trójki dotyczą tego samego wierzchołka. Gdy predykat reprezentuje krawędź, obiekt to wierzchołek (tak jak w trójce `_:idaho :within _:usa`). Jeśli predykat to właściwość, obiekt to literal tekstowy (tak jak w trójce `_:usa :name "Stany Zjednoczone"`).

Wielokrotne podawanie tego samego podmiotu skutkuje powtórzeniami, jednak na szczęście można zastosować średniki, aby zapisać wiele informacji na temat tego samego podmiotu. Dzięki temu format *Turtle* jest wygodny i czytelny (zob. listing 2.7).

### Listing 2.7. Zwięźlejszy sposób zapisu danych z listingu 2.6

```
@prefix : <urn:example:>.
_:lucy a :Person; :name "Lucy"; :bornIn _:idaho.
_:idaho a :Location; :name "Idaho"; :type "state"; :within _:usa.
_:usa a :Location; :name "Stany Zjednoczone"; :type "country"; :within _:namerica.
_:namerica a :Location; :name "Ameryka Północna"; :type "continent".
```

## Sieć semantyczna

Jeśli zaczniesz czytać o modelu triplestore, możesz się pogрузić w zalewie artykułów na temat *sieci semantycznej*. Model triplestore jest w pełni niezależny od sieci semantycznej. Na przykład Datomic [40] to baza używająca modelu triplestore, której twórcy nie piszą nic o powiązaniach z siecią semantyczną<sup>7</sup>. Jednak ponieważ w umysłach wielu osób te dwa rozwiązania są tak blisko ze sobą powiązane, należy pokrótce omówić sieć semantyczną.

Sieć semantyczna to prosty i sensowny pomysł. Witryny i tak obejmują informacje w postaci tekstów i grafiki przeznaczonych dla ludzi, dlaczego więc nie zamieszczać informacji w postaci danych czytelnych dla maszyn? Model **RDF** (ang. *Resource Description Framework*) [41] został pomyślany jako mechanizm do publikowania w różnych witrynach danych w spójnym formacie. Dzięki temu dane z rozmaitych serwisów można automatycznie łączyć w *sieć danych* — coś w rodzaju internetowej „bazy wszystkiego”.

Niestety, choć sieci semantycznej na początku obecnego wieku towarzyszył bardzo duży rozgłos, do tej pory brakuje oznak wykorzystania jej w praktyce, dlatego wiele osób sceptycznie odnosi się do tego podejścia. Problemem okazały się też ogrom akronimów, nadmiernie skomplikowane propozycje standardów i arogancja.

Jeśli jednak uda Ci się wyjść poza te niepowodzenia, zobaczysz, że z projektu sieci semantycznej powstało też wiele dobrych rzeczy. Trójki mogą być dobrym wewnętrznym modelem danych w aplikacjach nawet wtedy, jeśli nie zamierzasz publikować danych w modelu RDF w sieci semantycznej.

## Model danych RDF

Używany na listingu 2.7 język Turtle to czytelny dla człowieka format danych w modelu RDF. Model RDF zapisuje się też czasem w formacie XML, który pozwala uzyskać ten sam efekt, ale w dużo bardziej rozwlekły sposób (zob. listing 2.8). Język Turtle/N3 jest preferowany jako dużo czytelniejszy, a narzędzia takie jak Apache Jena [42] potrafią w razie potrzeby automatycznie przekształcać dane między różnymi formatami zgodnymi z RDF-em.

### Listing 2.8. Dane z listingu 2.7 zapisane z użyciem modelu RDF i języka XML

```
<rdf:RDF xmlns="urn:example:"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <Location rdf:nodeID="idaho">
    <name>Idaho</name>
    <type>state</type>
    <within>
```

---

<sup>7</sup> Technicznie Datomic używa krotek pięcioelementowych zamiast trójek (dwa dodatkowe pola zawierają metadane używane do kontroli wersji).

```

<Location rdf:nodeID="usa">
  <name>Stany Zjednoczone</name>
  <type>country</type>
  <within>
    <Location rdf:nodeID="america">
      <name>Ameryka Północna</name>
      <type>continent</type>
    </Location>
  </within>
</Location>
</within>
</Location>

<Person rdf:nodeID="lucy">
  <name>Lucy</name>
  <bornIn rdf:nodeID="idaho"/>
</Person>
</rdf:RDF>

```

RDF ma kilka dziwnych cech wynikających z tego, że został zaprojektowany do wymiany danych w internecie. Podmiot, predykat i obiekt w trójce to często identyfikatory URI. Na przykład predykatem może być URI `<http://my-company.com/namespace#within>` lub `<http://my-company.com/namespace#lives_in>`, a nie samo `WITHIN` lub `LIVES_IN`. Wynika to z tego, że możliwe powinno być łączenie Twoich danych z danymi innych ludzi, a jeśli przypiszą oni inne znaczenie słowom `within` lub `lives_in`, nie wystąpi konflikt, ponieważ predykaty będą miały postać `<http://other.org/foo#within>` i `<http://other.org/foo#lives_in>`.

Adres URL `<http://my-company.com/namespace>` nie musi do niczego prowadzić. W kontekście modelu RDF jest to po prostu przestrzeń nazw. Aby uniknąć potencjalnych pomyłek z adresami URL `http://`, w przykładach w tym punkcie używane są nieinterpretowalne identyfikatory URI typu `urn:example:within`. Na szczęście możesz podać ten przedrostek tylko raz na początku pliku, a potem o nim zapomnieć.

## Język zapytań SPARQL

SPARQL to język zapytań dla baz triplestore używających modelu danych RDF [43] (SPARQL — wymawiany „sparkel” — to akronim od nazwy *SPARQL Protocol and RDF Query Language*). Jest on starszy od języka Cypher, a ponieważ dopasowywanie wzorców w Cypherze zapożyczono z języka SPARQL, języki te wyglądają podobnie [37].

To samo zapytanie co wcześniej (wyszukiwanie osób, które urodziły się w Stanach Zjednoczonych przeprowadziły się do Europy) jest w języku SPARQL jeszcze bardziej zwarte niż w Cypherze (zob. listing 2.9).

Listing 2.9. To samo zapytanie co na listingu 2.4 zapisane w języku SPARQL

```

PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn / :within* / :name "Stany Zjednoczone".
  ?person :livesIn / :within* / :name "Europa".
}

```



Ta struktura wygląda znajomo. Dwa pokazane poniżej wyrażenia oznaczają to samo (w języku SPARQL zmienne zaczynają się od znaku zapytania).

```
(person) -[:BORN IN]-> () -[:WITHIN*0..]-> (location) # Cypher
?person :bornIn / :within* ?location. # SPARQL
```

Ponieważ w RDF-ie nie ma rozróżnienia na właściwości i krawędzie (dla obu używane są predykaty), do dostosowywania właściwości można się posłużyć tą samą składnią. W następnym wyrażeniu zmienna `usa` jest wiązana z każdym wierzchołkiem o właściwości `name` mającej wartość "Stany Zjednoczone":

```
(usa {name:'Stany Zjednoczone'}) # Cypher
?usa :name "Stany Zjednoczone". # SPARQL
```

SPARQL to wygodny język zapytań. Nawet jeśli sieć semantyczna nigdy nie stanie się rzeczywistością, ten język może się okazać wartościowym narzędziem do wewnętrznego użytku w aplikacjach.

## Bazy oparte na grafach a model sieciowy

W punkcie „Czy bazy oparte na dokumentach to powtórka z historii?” opisano, że CODASYL i model relacyjny współzawodniczyły w rozwiązywaniu problemu relacji wiele do wielu z baz IMS. Pozornie model sieciowy CODASYL jest podobny do modelu opartego na grafach. Czy bazy oparte na grafach to drugie wcielenie baz CODASYL w przebraniu?

Nie. Te rozwiązania różnią się między sobą pod kilkoma ważnymi względami:

- W modelu CODASYL baza ma schemat określający, który typ rekordowy można zagrzeżdżać w innym typie rekordowym. W bazach opartych na grafach takie ograniczenie nie występuje. Każdy wierzchołek może mieć krawędź do dowolnego innego wierzchołka. Zapewnia to aplikacjom znacznie większą swobodę w zakresie dostosowywania się do zmiennych wymagań.
- W modelu CODASYL jedyny sposób na dotarcie do konkretnego rekordu to poruszanie się jedną ze ścieżek dostępu. W bazie opartej na grafach można wskazać dowolny wierzchołek za pomocą unikatowego identyfikatora. Inna możliwość to wyszukiwanie wierzchołków o określonej wartości.
- W modelu CODASYL elementy podrzędne rekordu to uporządkowany zbiór, dlatego baza musi zachowywać kolejność elementów (co ma wpływ na układ przechowywanych danych), a aplikacja wstawiająca nowe rekordy do bazy musi uwzględniać pozycje nowych rekordów w zbiorach. W bazach opartych na grafach wierzchołki i krawędzie nie są uporządkowane (można jedynie posortować wyniki w ramach zapytania).
- W modelu CODASYL wszystkie zapytania są imperatywne i trudne do napisania. Ponadto łatwo można naruszyć ich działanie z powodu zmian w schemacie. W bazach opartych na grafach możesz (jeśli chcesz) pisać instrukcje poruszania się w kodzie imperatywnym, jednak większość takich baz udostępnia też wysokopoziomowe deklaratywne języki zapytań takie jak Cypher lub SPARQL.

## Podstawy — Datalog

*Datalog* to język znacznie starszy niż SPARQL lub Cypher. Został on dobrze przeanalizowany przez naukowców w latach 80. [44, 45, 46]. Wśród inżynierów oprogramowania jest mniej znany, jednak mimo to ma duże znaczenie, ponieważ zapewnia podstawy, na których zostały oparte późniejsze języki zapytań.

W praktyce Datalog wykorzystuje się w kilku systemach danych. Na przykład jako język zapytań w systemie Datomic [40], a Cascalog [47] to implementacja Datalogu używana w zapytaniach o duże zbiory danych w Hadoopie<sup>8</sup>.

Model danych w Datalogu przypomina uogólniony model triplestore. Zamiast zapisywać trójki w postaci (*podmiot*, *predykat*, *obiekt*), mają one format *predykat*(*podmiot*, *obiekt*). Na listingu 2.10 pokazano, jak zapisać przykładowe dane w Datalogu.

Listing 2.10. Podzbiór danych z rysunku 2.5 przedstawiony jako fakty w Datalogu

```
name(america, 'Ameryka Północna').
type(america, continent).

name(usa, 'Stany Zjednoczone').
type(usa, country).
within(usa, america).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

Po zdefiniowaniu danych można napisać to samo zapytanie co wcześniej — przedstawia to listing 2.11. To zapytanie wygląda nieco inaczej niż jego odpowiednik w językach Cypher lub SPARQL, jednak niech Cię to nie zniechęca. Datalog to podzbiór Prologu, z którym mogłeś się zetknąć, jeśli studiowałeś informatykę.

Listing 2.11. To samo zapytanie co na listingu 2.4 zapisane w Datalogu

```
within_recursive(Location, Name) :- name(Location, Name). /* Reguła 1 */

within_recursive(Location, Name) :- within(Location, Via), /* Reguła 2 */
                                     within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name), /* Reguła 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'Stany Zjednoczone', 'Europa').
/* Who = 'Lucy'. */
```

W językach Cypher i SPARQL od razu używana jest instrukcja SELECT, jednak w Datalogu robione są małe kroki. Definiowane są *reguły*, które informują bazę o nowych predykatkach. Tu zdefiniowano dwa nowe predykaty: *within\_recursive* i *migrated*. Te predykaty nie są przechowywanymi w bazie trójkami; tworzy się je na podstawie danych lub innych reguł. W regułach można stosować inne reguły (podobnie jak funkcje mogą wywoływać inne funkcje lub rekurencyjnie same siebie). W ten sposób można krok po kroku budować złożone zapytania.

---

<sup>8</sup> Datomic i Cascalog używają w Datalogu składni S-wyrażeń z języka Closure. W dalszych przykładach używana jest składnia języka Prolog (jest ona czytelniejsza), co jednak nie zmienia funkcjonowania kodu.

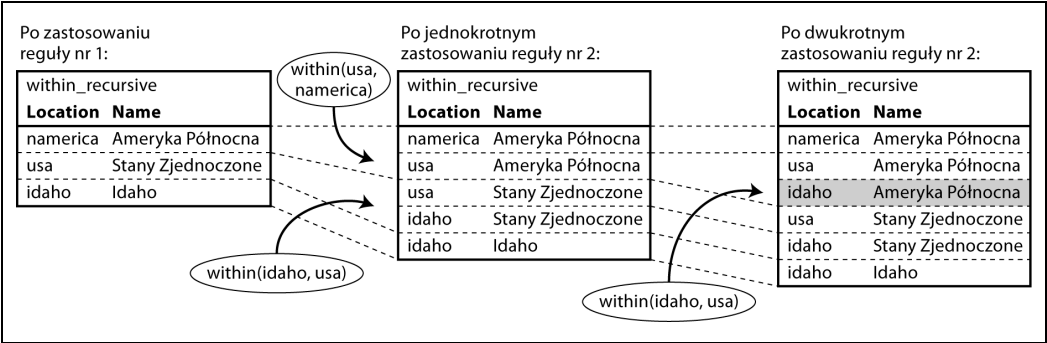
W regułach słowa zaczynające się wielką literą to zmienne, a predykaty są dopasowywane tak jak w językach Cypher i SPARQL. Na przykład `name(Location, Name)` pasuje do trójki `name(america, 'Ameryka Północna')`, gdzie wartości zmiennych to `Location = america` i `Name = 'Ameryka Północna'`.

Reguła zostaje zastosowana, jeśli system potrafi dopasować *wszystkie* predykaty podane po prawej stronie operatora `:-`. Zastosowanie reguły odpowiada dodaniu do bazy danych po lewej stronie operatora `:-` (przy czym zmienne są zastępowane dopasowanymi wartościami).

Oto możliwy sposób zastosowania reguł:

1. Dane `name(america, 'Ameryka Północna')` występują w bazie, więc reguła nr 1 zostaje zastosowana. Powoduje to wygenerowanie predykatu `within_recursive(america, 'Ameryka Północna')`.
2. Dane `within(usa, america)` występują w bazie, a we wcześniejszym kroku wygenerowano predykat `within_recursive(america, 'Ameryka Północna')`, dlatego reguła nr 2 zostaje zastosowana. Powoduje to wygenerowanie predykatu `within_recursive(usa, 'Ameryka Północna')`.
3. Dane `within(idaho, usa)` występują w bazie, a we wcześniejszym kroku wygenerowano predykat `within_recursive(usa, 'Ameryka Północna')`, dlatego reguła nr 2 zostaje zastosowana. Powoduje to wygenerowanie predykatu `within_recursive(idaho, 'Ameryka Północna')`.

Dzięki wielokrotnemu zastosowaniu reguł nr 1 i 2 predykat `within_recursive` pozwala ustalić wszystkie lokalizacje z Ameryki Północnej (lub lokalizacje o innej nazwie) zapisane w bazie danych. Ten proces przedstawiono na rysunku 2.6.



Rysunek 2.6. Używanie reguł Datalogu z listingu 2.11 do ustalenia, że Idaho znajduje się w Ameryce Północnej

Teraz reguła nr 3 pozwala znaleźć osoby urodzone w pewnej lokalizacji `BornIn` i mieszkające w jakiejś lokalizacji `LivingIn`. Użycie w zapytaniu wartości `BornIn = 'Stany Zjednoczone'` i `LivingIn = 'Europa'` oraz zapisanie osoby w zmiennej `Who` to żądanie do używającego Datalogu systemu, aby ustalił, jakie wartości może przyjąć zmienna `Who`. Ostatecznie uzyskiwana jest więc ta sama odpowiedź co we wcześniejszych zapytaniach w językach Cypher i SPARQL.

Podejście używane w Datalogu wymaga innego rodzaju myślenia niż w pozostałych językach zapytań omawianych w tym rozdziale. Daje jednak bardzo duże możliwości, ponieważ reguły można łączyć i ponownie wykorzystywać w innych zapytaniach. Jest mniej wygodne do tworzenia prostych jednorazowych zapytań, jednak pozwala uzyskać lepsze wyniki, jeśli dane są złożone.

# Podsumowanie

Modele danych to obszerny temat. W tym rozdziale pokrótce opisano wiele różnych modeli. Zabrakło miejsca na szczegółowe omawianie każdego modelu, mam jednak nadzieję, że ten przegląd wystarczył, aby zaostrzyć Twój apetyt na dokładniejsze zapoznanie się z modelem, który najlepiej pasuje do wymogów rozwijanej przez Ciebie aplikacji.

W przeszłości dane początkowo były reprezentowane jako jedno wielkie drzewo (model hierarchiczny). Nie było to jednak korzystne, by reprezentować relację wiele do wielu, dlatego w celu rozwiązania tego problemu wymyślono model relacyjny. Później programiści odkryli, że w niektórych zastosowaniach także model relacyjny się nie sprawdza. Nowe nierelacyjne magazyny danych typu NoSQL są rozwijane w dwóch kierunkach:

1. *Bazy oparte na dokumentach* są przeznaczone do przypadków użycia, w których dane znajdują się w niezależnych dokumentach, a relacje między jednym dokumentem a innymi są rzadkie.
2. *Twórcy baz opartych na grafach* poszli w przeciwnym kierunku i skupili się na przypadkach użycia, w których wszystko może być powiązane ze wszystkim.

Wszystkie trzy modele (oparty na dokumentach, relacyjny i oparty na grafach) są obecnie powszechnie używane, a każdy z nich jest przydatny w odpowiadającej mu dziedzinie. Jeden model można przedstawić w kategoriach innego modelu (np. dane z grafu można zapisać w bazie relacyjnej), jednak takie rozwiązanie bywa często niewygodne. To dlatego zamiast jednego uniwersalnego rozwiązania istnieją różne systemy przeznaczone do różnych celów.

Cechę wspólną baz opartych na dokumentach i grafach stanowi to, że zwykle nie narzucają one schematu przechowywanych danych, co ułatwia dostosowanie aplikacji do zmieniających się wymogów. Jednak w aplikacji i tak zwykle się przyjmuje, że dane mają określoną strukturę. Pozostaje kwestia tego, czy schemat jest określany bezpośrednio (wymuszany przy zapisie), czy pośrednio (obsługiwany przy odczycie).

Każdy model danych ma odrębny język lub platformę zapytań. Tu opisano kilka przykładów: SQL, MapReduce, potoki agregacji z baz MongoDB, Cypher, SPARQL i Datalog. Wspomniano też o językach CSS i XSL/XPath, które nie są językami zapytań dla baz danych, ale występują w nich ciekawe analogie.

Choć opisano tu wiele zagadnień, liczne modele danych nie zostały wymienione. Oto kilka krótkich przykładów:

- Badacze pracujący nad danymi dotyczącymi genomu często przeprowadzają *poszukiwanie podobieństwa sekwencji*. Oznacza to, że biorą jeden bardzo długi łańcuch znaków (reprezentujący cząsteczkę DNA) i dopasowują go do dużej bazy podobnych, ale nie identycznych łańcuchów znaków. Żadna z opisanych tu baz nie obsługuje tego rodzaju zastosowań. Dlatego badacze opracowali wyspecjalizowane oprogramowanie dla baz przechowujących genom takie jak GenBank [48].
- Fizycy cząstek elementarnych od dziesięcioleci przeprowadzają rozbudowane analizy danych metodami podobnymi jak w Big Data. W projektach takich jak Wielki Zderzacz Hadronów używane są obecnie setki petabajtów! Przy tej skali niezbędne są niestandardowe rozwiązania, aby koszty sprzętu nie wymknęły się spod kontroli [49].

- Wyszukiwanie pełnotekstowe to zdaniem niektórych osób model danych często używany razem z bazami. Pobieranie informacji to obszerny i specjalistyczny temat, który nie został szczegółowo omówiony w tej książce. Jednak w rozdziale 3. i części III pokrótce opisano indeksy wyszukiwania.

Na razie musimy pozostawić te zagadnienia. W następnym rozdziale omówiono kompromisy, które pojawiają się w trakcie *implementowania* modeli danych opisanych w tym rozdziale.

## **Literatura cytowana**

- [1] Edgar F. Codd, *A Relational Model of Data for Large Shared Data Banks*, „Communications of the ACM”, rocznik 13, nr 6, s. 377 – 387, czerwiec 1970 (<https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>; <https://dl.acm.org/citation.cfm?doid=362384.362685>).
- [2] Michael Stonebraker i Joseph M. Hellerstein, *What Goes Around Comes Around*, w: „Readings in Database Systems, 4th edition”, MIT Press, s. 2 – 41, 2005, ISBN: 978-0-262-69314-1 (<http://mitpress2.mit.edu/books/chapters/0262693143chapm1.pdf>).
- [3] Pramod J. Sadalage i Martin Fowler, *NoSQL Distilled*, Addison-Wesley, sierpień 2012, ISBN: 978-0-321-82662-6.
- [4] Eric Evans, *NoSQL: What's in a Name?*, [blog.sym-link.com](http://blog.sym-link.com), 30 października 2009 ([http://blog.sym-link.com/2009/10/30/nosql\\_whats\\_in\\_a\\_name.html](http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html)).
- [5] James Phillips, *Surprises in Our NoSQL Adoption Survey*, [blog.couchbase.com](http://blog.couchbase.com), 8 lutego 2012 (<https://blog.couchbase.com/nosql-adoption-survey-surprises/>).
- [6] Michael Wagner, *SQL/XML:2006 — Evaluierung der Standardkonformität ausgewählter Datenbanksysteme*. Diplomica Verlag, Hamburg, 2010, ISBN: 978-3-836-64609-3.
- [7] *XML Data in SQL Server*, dokumentacja systemu SQL Server 2012, [technet.microsoft.com](http://technet.microsoft.com), 2013 (<https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server>).
- [8] *PostgreSQL 9.3.1 Documentation*, The PostgreSQL Global Development Group, 2013 (<https://www.postgresql.org/docs/9.3/static/index.html>).
- [9] *The MongoDB 2.4 Manual*, MongoDB, Inc., 2013 (<https://docs.mongodb.com/manual/>).
- [10] *RethinkDB 1.11 Documentation*, [rethinkdb.com](http://rethinkdb.com), 2013 (<https://rethinkdb.com/docs/>).
- [11] *Apache CouchDB 1.6 Documentation*, [docs.couchdb.org](http://docs.couchdb.org), 2014 (<http://docs.couchdb.org/en/latest/>).
- [12] Lin Qiao, Kapil Surlaker, Shirshanka Das i in., *On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2013 (<https://www.slideshare.net/amywtang/espresso-20952131>).
- [13] Rick Long, Mark Harrington, Robert Hain i Geoff Nicholls, *IMS Primer*, IBM Redbook SG24-5352-00, IBM International Technical Support Organization, styczeń 2000 (<http://www.redbooks.ibm.com/redbooks/pdfs/sg245352.pdf>).

- [14] Stephen D. Bartlett, *IBM's IMS — Myths, Realities, and Opportunities*, „The Clipper Group Navigator”, TCG2013015LI, lipiec 2013 (<ftp://public.dhe.ibm.com/software/data/ims/pdf/TCG2013015LI.pdf>).
- [15] Sarah Mei, *Why You Should Never Use MongoDB*, [sarahmei.com](http://www.sarahmei.com), 11 listopada 2013 (<http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/>).
- [16] J.S. Knowles i D.M.R. Bell, *The CODASYL Model*, „Databases — Role and Structure: An Advanced” Course, red. P.M. Stocker, P.M.D. Gray i M.P. Atkinson, s. 19 – 56, Cambridge University Press, 1984, ISBN: 978-0-521-25430-4.
- [17] Charles W. Bachman, *The Programmer as Navigator*, „Communications of the ACM”, rocznik 16, nr 11, s. 653 – 658, listopad 1973 (<https://dl.acm.org/citation.cfm?id=362534>; <https://dl.acm.org/citation.cfm?doid=355611.362534>).
- [18] Joseph M. Hellerstein, Michael Stonebraker i James Hamilton, *Architecture of a Database System*, „Foundations and Trends in Databases”, rocznik 1, nr 2, s. 141 – 259, listopad 2007 (<http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf>; <http://www.nowpublishers.com/article/Details/DBS-002>).
- [19] Sandeep Parikh i Kelly Stirman, *Schema Design for Time Series Data in MongoDB*, [blog.mongodb.org](http://blog.mongodb.org), 30 października 2013 (<https://www.mongodb.com/blog/post/schema-design-for-time-series-data-in-mongodb>).
- [20] Martin Fowler, *Schemaless Data Structures*, [martinfowler.com](http://martinfowler.com), 7 stycznia 2013 (<https://martinfowler.com/articles/schemaless/>).
- [21] Amr Awadallah, *Schema-on-Read vs. Schema-on-Write*, w: „Berkeley EECS RAD Lab Retreat”, Santa Cruz, CA, maj 2009 (<https://www.slideshare.net/awadallah/schemaonread-vs-schemaonwrite>).
- [22] Martin Odersky, *The Trouble with Types*, w: „Strange Loop”, wrzesień 2013 (<https://www.infoq.com/presentations/data-types-issues>).
- [23] Conrad Irwin, *MongoDB — Confessions of a PostgreSQL Lover*, w: „HTML5DevConf”, październik 2013 (<https://speakerdeck.com/conradirwin/mongodb-confessions-of-a-postgresql-lover>).
- [24] *Percona Toolkit Documentation: pt-online-schema-change*, Percona Ireland Ltd., 2013 (<https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html>).
- [25] Rany Keddo, Tobias Bielohlawek i Tobias Schmidt, *Large Hadron Migrator*, SoundCloud, 2013 (<https://github.com/soundcloud/lhm>).
- [26] Shlomi Noach, *gh-ost: GitHub's Online Schema Migration Tool for MySQL*, [githubengineering.com](http://githubengineering.com), 1 sierpnia 2016 (<https://githubengineering.com/gh-ost-github-s-online-migration-tool-for-mysql/>).
- [27] James C. Corbett, Jeffrey Dean, Michael Epstein i in., *Spanner: Google's Globally-Distributed Database*, w: *10th USENIX Symposium on Operating System Design and Implementation (OSDI)*, październik 2012 (<https://research.google.com/archive/spanner.html>).
- [28] Donald K. Burleson, *Reduce I/O with Oracle Cluster Tables*, [dba-oracle.com](http://dba-oracle.com) ([http://www.dba-oracle.com/oracle\\_tip\\_hash\\_index\\_cluster\\_table.htm](http://www.dba-oracle.com/oracle_tip_hash_index_cluster_table.htm)).

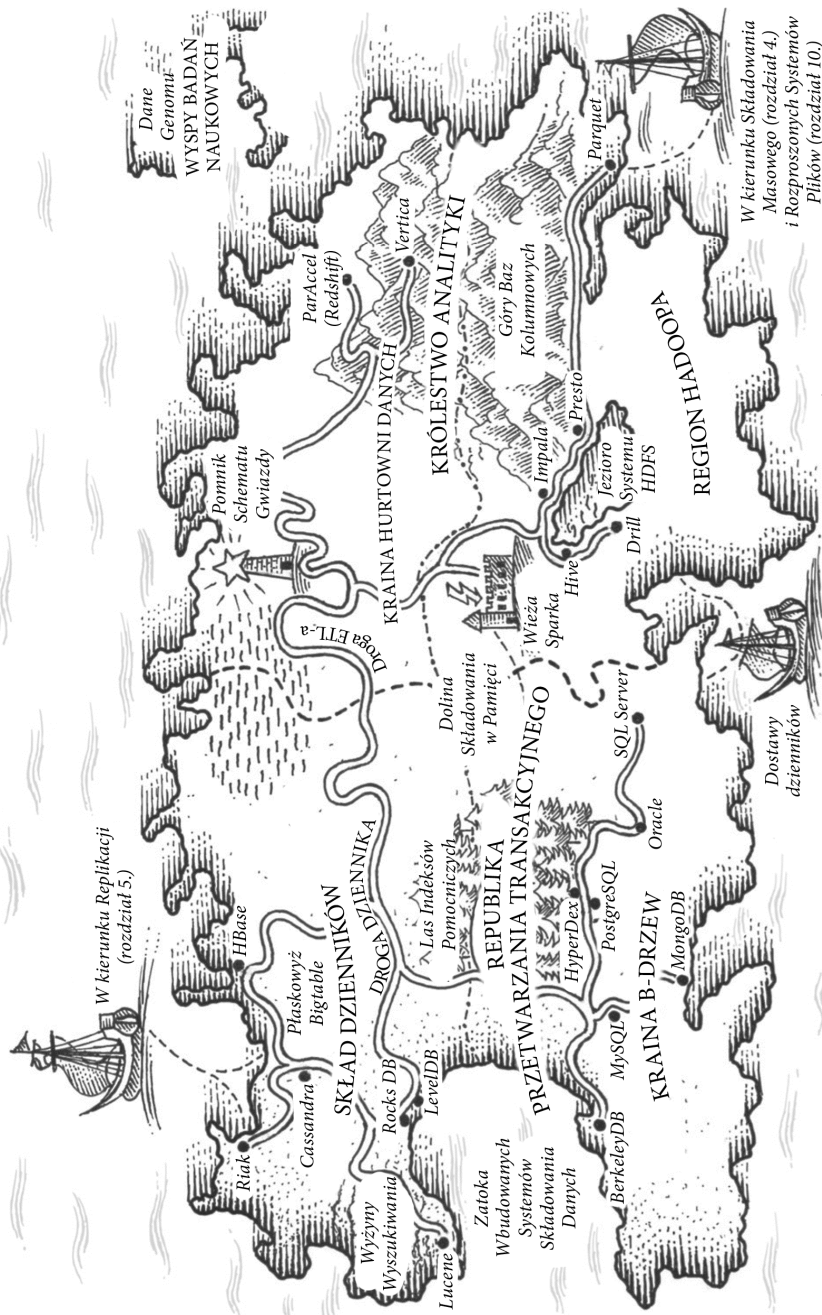
- [29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat i in., *Bigtable: A Distributed Storage System for Structured Data*, w: *7th USENIX Symposium on Operating System Design and Implementation (OSDI)*, listopad 2006 (<https://research.google.com/archive/bigtable.html>).
- [30] Bobbie J. Cochrane i Kathy A. McKnight, *DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON*, „IBM developerWorks”, 20 czerwca 2013 (<https://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson1/>).
- [31] Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, „Dr. Dobb’s Journal”, rocznik 30, nr 3, s. 202 – 210, marzec 2005 (<http://www.gotw.ca/publications/concurrency-ddj.htm>).
- [32] Joseph M. Hellerstein, *The Declarative Imperative: Experiences and Conjectures in Distributed Logic*, Electrical Engineering and Computer Sciences, University of California at Berkeley, raport techniczny UCB/EECS-2010-90, czerwiec 2010 (<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-90.pdf>).
- [33] Jeffrey Dean i Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, w: *6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, grudzień 2004 (<https://research.google.com/archive/mapreduce.html>).
- [34] Craig Kerstiens, *JavaScript in Your Postgres*, *blog.heroku.com*, 5 czerwca 2013 ([https://blog.heroku.com/javascript\\_in\\_your\\_postgres](https://blog.heroku.com/javascript_in_your_postgres)).
- [35] Nathan Bronson, Zach Amsden, George Cabrera i in., *TAO: Facebook’s Distributed Data Store for the Social Graph*, w: „USENIX Annual Technical Conference” (USENIX ATC), czerwiec 2013 (<https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson>).
- [36] *Apache TinkerPop3.2.3 Documentation*, *tinkerpop.apache.org*, październik 2016 (<http://tinkerpop.apache.org/docs/3.2.3/reference/>).
- [37] *The Neo4j Manual v2.0.0*, „Neo Technology”, 2013 (<http://neo4j.com/docs/2.0.0/index.html>).
- [38] Emil Eifrem, korespondencja z Twittera, 3 stycznia 2014 (<https://twitter.com/emileifrem/status/419107961512804352>).
- [39] David Beckett i Tim Berners-Lee, *Turtle — Terse RDF Triple Language*, „W3C Team Submission”, 28 marca 2011 (<https://www.w3.org/TeamSubmission/turtle/>).
- [40] *Datomic Development Resources*, „Metadata Partners”, LLC, 2013 (<http://docs.datomic.com/>).
- [41] W3C RDF Working Group, *Resource Description Framework (RDF)*, *w3.org*, 10 lutego 2004 (<https://www.w3.org/RDF/>).
- [42] *Apache Jena*, Apache Software Foundation (<http://jena.apache.org/>).
- [43] Steve Harris, Andy Seaborne i Eric Prud’hommeaux, *SPARQL 1.1 Query Language*, „W3C Recommendation”, marzec 2013 (<https://www.w3.org/TR/sparql11-query/>).
- [44] Todd J. Green, Shan Shan Huang, Boon Thau Loo i Wenchao Zhou, *Datalog and Recursive Query Processing*, „Foundations and Trends in Databases”, rocznik 5, nr 2, s. 105 – 195, listopad 2013 (<http://blogs.evergreen.edu/sosw/files/2014/04/Green-Vol5-DBS-017.pdf>; <http://www.nowpublishers.com/article/Details/DBS-017>).

- [45] Stefano Ceri, Georg Gottlob i Letizia Tanca, *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*, „IEEE Transactions on Knowledge and Data Engineering”, rocznik 1, nr 1, s. 146 – 166, marzec 1989 ([https://www.researchgate.net/profile/Letizia\\_Tanca/publication/3296132\\_What\\_you\\_always\\_wanted\\_to\\_know\\_about\\_Datalog\\_and\\_never\\_dared\\_to\\_ask/links/0fcfd50ca2d20473ca000000.pdf](https://www.researchgate.net/profile/Letizia_Tanca/publication/3296132_What_you_always_wanted_to_know_about_Datalog_and_never_dared_to_ask/links/0fcfd50ca2d20473ca000000.pdf); <http://ieeexplore.ieee.org/document/43410/>).
- [46] Serge Abiteboul, Richard Hull i Victor Vianu, *Foundations of Databases*, Addison-Wesley, 1995, ISBN: 978-0-201-53771-0, dostępne w internecie na stronie <http://webdam.inria.fr/Alice/>.
- [47] Nathan Marz, *Casalog*, [casalog.org](http://casalog.org/) (<http://casalog.org/>).
- [48] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman i in., *GenBank*, „Nucleic Acids Research”, rocznik 36, nr poświęcony bazom danych, s. D25 – D30, grudzień 2007 ([https://academic.oup.com/nar/article/36/suppl\\_1/D25/2507746/GenBank](https://academic.oup.com/nar/article/36/suppl_1/D25/2507746/GenBank); <https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkm929>).
- [49] Fons Rademakers, *ROOT for Big Data Analysis*, w: *Workshop on the Future of Big Data Management*, Londyn, czerwiec 2013 (<https://indico.cern.ch/event/246453/contributions/1566610/attachments/423154/587535/ROOT-BigData-Analysis-London-2013.pdf>).





## OCEAN DANYCH ROZPROSZONYCH



# Przechowywanie i pobieranie danych

*Wer Ordnung hält, ist nur zu faul zum Suchen.  
(Jeśli dbasz o porządek, oznacza to, że jesteś zbyt leniwy, aby szukać).*

— przysłowie niemieckie

Na najbardziej podstawowym poziomie baza musi robić dwie rzeczy: gdy przekażesz jej dane, powinna je przechowywać, a gdy później ich zażądasz, powinna je udostępnić.

W rozdziale 2. opisano modele danych i języki zapytań, czyli format, w jakim programista aplikacji przekazuje dane bazie, oraz mechanizm, który pozwala ich później zażądać. W tym rozdziale te same zagadnienia są przedstawione z perspektywy bazy danych. Zobaczysz, jak można zapisywać otrzymane dane i jak je później odszukać, gdy użytkownik ich zażąda.

Dlaczego programista aplikacji ma dbać o to, jak baza wewnętrznie obsługuje przechowywanie i pobieranie danych? Prawdopodobnie nie zamierzasz od podstaw implementować systemu składowania danych, *musisz* jednak spośród wielu dostępnych rozwiązań wybrać to, które będzie odpowiednie do tworzonej aplikacji. Aby zapewnić dobrą wydajność systemu przy określonym obciążeniu roboczym, musisz mieć ogólne pojęcie na temat pracy wewnętrznych mechanizmów systemu składowania danych.

Występuje duża różnica między systemami składowania danych zoptymalizowanymi pod kątem transakcyjnego obciążenia roboczego i zoptymalizowanymi na potrzeby analityki. To rozróżnienie opisano dalej w punkcie „Przetwarzanie transakcji czy analityka?”, a w punkcie „Przechowywanie z użyciem kolumn” omówiono rodzinę systemów zoptymalizowanych pod kątem analityki.

Jednak ten rozdział zaczyna się od omówienia systemów używanych w bazach, które zapewne znasz (w bazach relacyjnych), a także w większości tzw. baz NoSQL. Opisane są tu dwie rodziny systemów składowania danych: *o strukturze dziennika* i *oparte na stronach* (np. z b-drzewami).

## Struktury danych używane w bazie

Przyjrzyj się najprostszej możliwej bazie zaimplementowanej jako dwie funkcje basha:

```
#!/bin/bash

db_set () {
    echo "$1,$2">> database
}
```

```
db_get () {
    grep "^$1,"database |sed -e "s/^$1,/" |tail -n 1
}
```

Te dwie funkcje obsługują magazyn par klucz-wartość. Możesz wywołać polecenie `db_set key value`, które zapisuje `key` i `value` w bazie. Kluczem i wartością może być (prawie) cokolwiek. Wartością może być np. dokument w formacie JSON. Następnie możesz wywołać polecenie `db_get key`, które wyszukuje najnowszą wartość powiązaną z danym kluczem i zwraca ją.

I to rozwiązanie działa:

```
$ db_set 123456 '{"name":"Londyn","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Format przechowywania danych jest bardzo prosty: używa się pliku tekstowego, w którym każdy wiersz obejmuje parę rozdzielonych przecinkiem elementów klucz-wartość (podobnie wygląda plik CSV, jeśli pominąć znaki ucieczki). Każde wywołanie `db_set` dodaje dane na koniec pliku. Dlatego jeśli zaktualizujesz klucz kilkakrotnie, starsze wersje wartości nie zostaną zastąpione. Aby znaleźć najnowszą wartość, trzeba sprawdzić ostatnie wystąpienie klucza w pliku (stąd instrukcja `tail -n 1` w funkcji `db_get`):

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}
```

```
$ cat database
123456,{"name":"Londyn","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Funkcja `db_set` ma dobrą wydajność jak na tak proste rozwiązanie. Dzieje się tak, ponieważ dodawanie danych to pliku jest zwykle wysoce wydajne. Wiele baz działa podobnie jak funkcja `db_set` i używa *dziennika*, do którego dane są wyłącznie dodawane. W rzeczywistych bazach trzeba sobie radzić także z innymi kwestiami (np. kontrolą współbieżności, odzyskiwaniem przestrzeni na dysku, aby dziennik nie rozrastał się w nieskończoność, i obsługą błędów oraz częściowo zapisanych rekordów), jednak podstawowa zasada pozostaje niezmienna. Dzienniki są bardzo przydatne i kilkakrotnie natrafisz na nie w dalszych częściach książki.



Słowo *dziennik* często oznacza dziennik aplikacji, gdzie aplikacja umieszcza tekst opisujący, co się dzieje. W tej książce *dziennik* ma ogólniejsze znaczenie — to sekwencja rekordów, do której można tylko dodawać elementy. Dziennik nie musi być czytelny dla człowieka. Może mieć postać binarną i być przeznaczony do odczytu tylko przez inne programy.

Natomiast jeśli liczba rekordów w bazie jest duża, wydajność funkcji `db_get` jest bardzo niska. Za każdym razem, gdy chcesz znaleźć klucz, funkcja `db_get` musi sprawdzić całą bazę od początku do końca i znaleźć wystąpienia tego klucza. W kategoriach algorytmicznych koszt wyszukiwania wynosi  $O(n)$ . Jeśli podwoisz liczbę rekordów w bazie ( $n$ ), wyszukiwanie potrwa dwa razy dłużej. Nie jest to korzystne.

Aby wydajnie wyszukiwać w bazie wartość odpowiadającą danemu kluczowi, potrzebna jest inna struktura danych — *indeks*. W tym rozdziale znajdziesz omówienie i porównanie różnych struktur używanych do indeksowania. Ogólnie chodzi o to, by osobno przechowywać dodatkowe metadane, które działają jak drogowskaz i pomagają znaleźć szukane dane. Jeśli chcesz przeszukiwać te same dane na kilka różnych sposobów, możliwe, że będziesz potrzebował kilku różnych indeksów dotyczących różnych części danych.

Indeks to *dodatkowa* struktura tworzona na podstawie podstawowych danych. Wiele baz umożliwia dodawanie i usuwanie indeksów, które nie wpływają na zawartość bazy, a wyłącznie na wydajność zapytań. Przechowywanie dodatkowych struktur powoduje dodatkowe koszty (zwłaszcza na etapie zapisu). W przypadku zapisu trudno jest poprawić wydajność prostej operacji dodawania danych do pliku, ponieważ to najprostszy możliwy sposób zapisu. Każdy indeks zwykle spowalnia zapis, ponieważ przy każdej takiej operacji konieczna jest też aktualizacja indeksu.

To ważny kompromis w systemie składowania danych — dobrze dobrane indeksy przyspieszają zapytania związane z odczytem, ale każdy indeks spowalnia zapis. Dlatego bazy zwykle nie indeksują domyślnie wszystkich danych, ale wymagają od programisty aplikacji lub administratora bazy, by ręcznie wybrał indeksy na podstawie wiedzy na temat typowych wzorców zapytań w aplikacji. Następnie możesz wybrać indeksy, które zapewniają aplikacji największe korzyści, nie powodując więcej kosztów niż to konieczne.

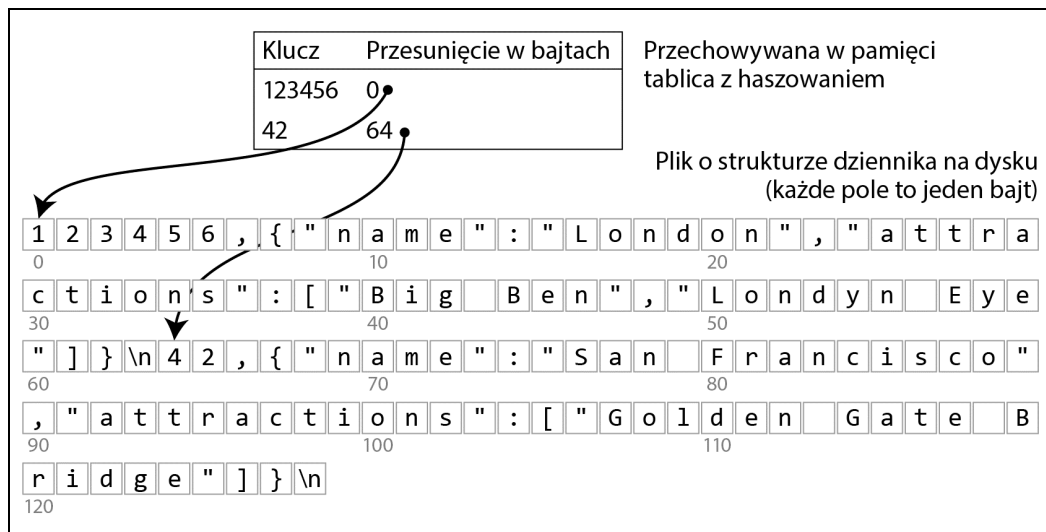
## Indeksy z haszowaniem

Zacznijmy od indeksów dla danych w postaci klucz-wartość. Nie jest to jedyny rodzaj indeksowanych danych, ale występuje bardzo często i stanowi przydatny komponent bardziej złożonych indeksów.

Magazyny kluczy i wartości są podobne do *słowników*, które znajdziesz w większości języków programowania. Zwykle takie magazyny są implementowane jako tablica z haszowaniem (ang. *hash table* lub *hash map*). Takie tablice są opisane w wielu podręcznikach poświęconych algorytmom [1, 2], dlatego ich działanie nie zostało tu szczegółowo omówione. Ponieważ istnieją już tablice z haszowaniem dla struktur danych przechowywanych w pamięci, dlaczego nie wykorzystasz ich do indeksowania danych na dysku?

Załóżmy, że dane są tylko dodawane do pliku (tak jak w opisanym wcześniej przykładzie). Wtedy najprostsza strategia indeksowania wygląda tak: przechowywać w pamięci tablicę z haszowaniem, w której każdemu kluczowi odpowiada przesunięcie w bajtach z pliku z danymi. To przesunięcie wskazuje miejsce, w którym można znaleźć wartość (zob. rysunek 3.1). Gdy dodajesz do pliku nową parę klucz-wartość, aktualizujesz też tablicę z haszowaniem, aby odzwierciedlić przesunięcie zapisanych danych (dotyczy to zarówno wstawiania nowych kluczy, jak i aktualizowania istniejących). Gdy chcesz znaleźć wartość, zastosuj tablicę z haszowaniem, aby ustalić przesunięcie w pliku z danymi, przejdź do określonej lokalizacji i wczytaj wartość.

Może się to wydawać nadmiernie uproszczone, a jednak to akceptowalne rozwiązanie. Mniej więcej tak działa Bitcask (domyślny system składowania danych w produktach Riak) [3]. Bitcask zapewnia wydajne odczyty i zapisy, przy czym wymóg jest taki, by wszystkie klucze mieściły się w dostępnej pamięci RAM, ponieważ tablica z haszowaniem jest w całości przechowywana w pamięci. Wartości mogą zajmować więcej miejsca niż jest dostępnej pamięci, ponieważ można je wczytać



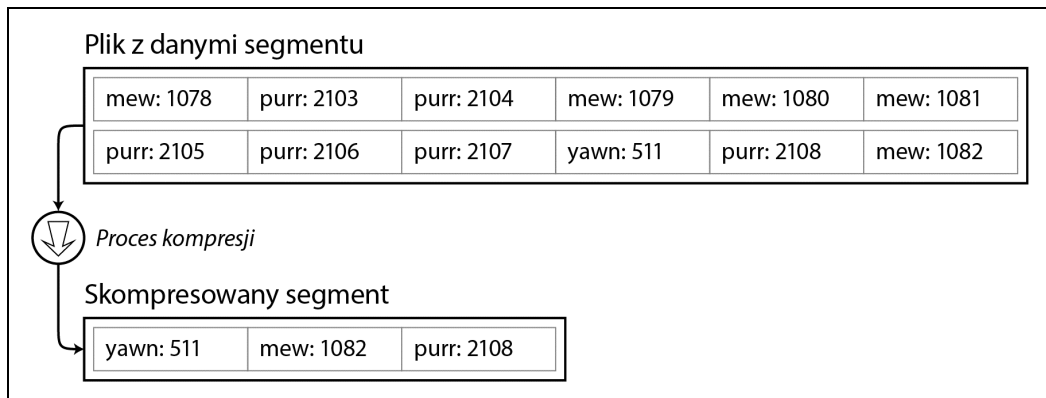
Rysunek 3.1. Przechowywanie dziennika z parami klucz-wartość w formacie podobnym do plików CSV. Dziennik jest indeksowany z użyciem przechowywanej w pamięci tablicy z haszowaniem

z dysku z użyciem tylko jednej operacji wyszukiwania. Jeśli potrzebna część pliku z danymi znajduje się już w pamięci podręcznej systemu plików, odczyt nie wymaga żadnych dyskowych operacji wejścia-wyjścia.

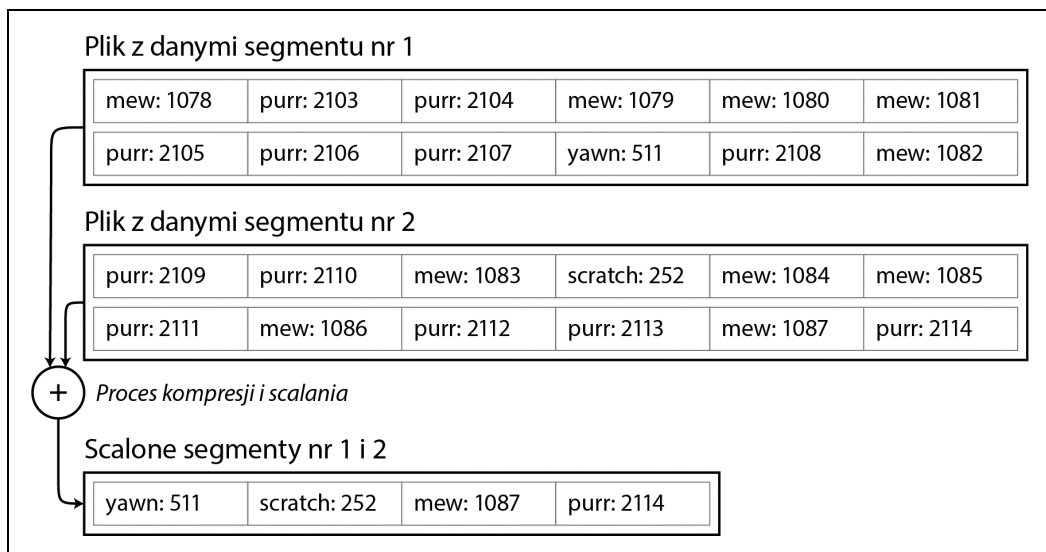
System składowania danych taki jak Bitcask jest dobrze dostosowany do sytuacji, w których wartość każdego klucza jest często aktualizowana. Kluczem może być np. adres URL filmu z kodem, a wartością liczba jego wyświetleń (zwiększana za każdym razem, gdy ktoś wciśnie przycisk odtwarzania). Obciążenie robocze tego rodzaju powoduje wiele zapisów, natomiast liczba różnych kluczy nie jest duża. Liczba zapisów dla każdego klucza jest wysoka, można przechowywać wszystkie klucze w pamięci.

Zgodnie z dotychczasowym opisem dane są wyłącznie dodawane do pliku. Jak więc uniknąć wyczerpania miejsca na dysku? Dobrze rozwiązanie stanowi podział dziennika na segmenty określonej wielkości. Należy wtedy zamykać plik z segmentem po osiągnięciu ustalonego rozmiaru i dokonywać dalszych zapisów w pliku nowego segmentu. Następnie można *skompresować* segmenty w sposób pokazany na rysunku 3.2. Kompresja polega na usunięciu powtarzających się kluczy z dziennika i zachowaniu tylko najnowszej wersji każdego klucza.

Ponieważ kompresja często sprawia, że segmenty stają się znacznie mniejsze (przy założeniu, że wartość klucza jest zmieniana średnio kilka razy na segment), w momencie kompresji można scalać parę segmentów, co przedstawia rysunek 3.3. Segmenty po zapisaniu nigdy nie są modyfikowane, dlatego scalony segment jest zachowywany w nowym pliku. Scalanie i kompresję zamrożonych segmentów da się wykonywać w wątku tła. W trakcie wykonywania tych operacji można obsługiwać żądania odczytu i zapisu w standardowy sposób, używając starszych plików z segmentami. Po zakończeniu scalania do obsługi żądań odczytu należy zacząć używać nowego scalonego segmentu zamiast starszych segmentów. Potem starsze pliki z segmentami można usunąć.



Rysunek 3.2. Kompresja dziennika par klucz-wartość (zliczającego odtworzenia każdego filmu z kotem). Dla każdego klucza zachowywana jest tylko najnowsza wartość



Rysunek 3.3. Jednoczesne kompresowanie i scalanie segmentów

Teraz dla każdego segmentu istnieje odrębna tablica z haszowaniem łącząca klucze z przesunięciami z pliku. Aby znaleźć powiązaną z kluczem wartość, najpierw należy sprawdzić tablicę dla najnowszego segmentu. Jeśli dany klucz w niej nie występuje, sprawdzany jest segment drugi w kolejności itd. Proces scalania sprawia, że liczba segmentów jest niewielka. Dzięki temu w procesie wyszukiwania nie trzeba sprawdzać wielu tablic.

Aby ten prosty pomysł zadziałał w praktyce, trzeba uwzględnić wiele szczegółów. Oto krótki przegląd zagadnień ważnych w rzeczywistych implementacjach:

#### Format pliku

CSV nie jest optymalnym formatem dziennika. Szybciej i prościej będzie wykorzystać format binarny, w którym najpierw zakodowano długość łańcucha znaków w bajtach, a następnie sam łańcuch (nie są wtedy potrzebne znaki ucieczki).

## Usuwanie rekordów

Jeśli chcesz usunąć klucz i powiązaną z nim wartość, musisz dodać do pliku z danymi specjalny rekord usuwania (ang. *tombstone*, czyli *nagrobek*). W trakcie scalania segmentów dziennika nagrobek jest informacją, by pominąć wszystkie wcześniejsze wartości usuniętego klucza.

## Przywracanie stanu po awarii

Gdy baza jest ponownie uruchamiana, przechowywane w pamięci tablice z haszowaniem zostają utracone. Teoretycznie można odzyskać tablicę dla każdego segmentu, odczytując cały plik segmentu od początku do końca i zapisując przy tym przesunięcie najnowszej wartości każdego klucza. Jednak jeśli pliki segmentu są długie, ten proces może zająć dużo czasu, przez co restartowanie serwera będzie kłopotliwe. Bitcask przyspiesza proces odzyskiwania, zapisując na dysku snapshoty tablicy z haszowaniem dla każdego segmentu. Te snapshoty można dość szybko wczytać do pamięci.

## Częściowo zapisane rekordy

Baza może w każdej chwili ulec awarii, także w trakcie dodawania rekordu do dziennika. Pliki w systemie Bitcask obejmują sumy kontrolne umożliwiające wykrywanie i ignorowanie uszkodzonych fragmentów dziennika.

## Kontrola współbieżności

Gdy dane są dodawane do dziennika ściśle sekwencyjnie, standardową implementacją jest używanie tylko jednego wątku zapisującego. Plik z danymi segmentu umożliwia tylko dodawanie danych (oprócz tego jest niemodyfikowalny), dzięki czemu wiele wątków może go równolegle odczytywać.

Dziennik umożliwiający jedynie zapis na pozór może wydawać się marnotrawstwem. Dlaczego nie aktualizować pliku od razu, zastępując dawną wartość nową? Jednak projekt pozwalający jedynie na dodawanie danych okazuje się przydatny z kilku przyczyn:

- Dołączanie danych i scalanie segmentów to operacje zapisu sekwencyjnego, które zwykle są znacznie szybsze niż zapis losowy (zwłaszcza na magnetycznych dyskach obrotowych). Zapis sekwencyjny jest też w pewnym stopniu preferowany w opartych na pamięci flash dyskach **SSD** (ang. *solid state drives*) [4]. To zagadnienie zostało opisane dokładniej w punkcie „Porównanie b-drzew i drzew LSM”.
- Obsługa współbieżności i przywracania stanu po awarii są znacznie prostsze, gdy pliki segmentów umożliwiają wyłącznie zapis lub są niemodyfikowalne. Nie musisz wtedy np. martwić się o wystąpienie awarii w momencie zastępowania wartości, kiedy to powstaje plik zawierający fragment starych i fragment nowych danych połączonych ze sobą.
- Scalanie dawnych segmentów pozwala uniknąć problemu fragmentacji plików z danymi w czasie.

Indeks oparty na tablicy z haszowaniem ma jednak pewne ograniczenia:

- Tablica z haszowaniem musi się mieścić w pamięci. Dlatego gdy liczba kluczy jest bardzo duża, pojawia się problem. Teoretycznie można przechowywać tablicę z haszowaniem na dysku, jednak niestety trudno jest zapewnić wysoką wydajność takiego rozwiązania. Wymaga ono



wielu operacji wejścia-wyjścia z dostępem losowym, powiększanie pełnej tablicy jest kosztowne, a obsługa kolizji wiąże się ze skomplikowaną logiką [5].

- Zapytania z wykorzystaniem przedziałów są wtedy niewygodne. Nie można np. łatwo sprawdzić wszystkich kluczy z przedziału od `kitty00000` do `kitty99999`. Trzeba sprawdzić każdy klucz z osobna w tablicach z haszowaniem.

W następnym punkcie opisano używaną do indeksowania strukturę pozbawioną takich ograniczeń.

## Pliki SSTable i drzewa LSM

Na rysunku 3.3 każdy przechowujący dane segment o strukturze dziennika jest sekwencją par klucz-wartość. Te pary pojawiają się w kolejności, w jakiej zostały zapisane, a wartości umieszczone dalej w dzienniku są ważniejsze niż wcześniejsze wartości powiązane z tym samym kluczem. Oprócz tego kolejność par klucz-wartość w pliku nie ma znaczenia.

Teraz wprowadźmy prostą zmianę w formacie plików z segmentami i dodajmy wymóg, by sekwencja par klucz-wartość była *posortowana według klucza*. Ten wymóg na pozór uniemożliwia sekwencyjny zapis. Niedługo wrócimy do tego zagadnienia.

Ten format to *SSTable* (ang. *Sorted String Table*). Inny wymóg stanowi to, by każdy klucz występował w każdym scalonym pliku segmentu tylko raz (proces kompresji już to gwarantuje). Pliki SSTable mają kilka ważnych zalet w porównaniu z segmentami dziennika z indeksami opartymi na haszowaniu:

1. Scalanie segmentów jest proste i wydajne nawet wtedy, gdy pliki są większe niż ilość dostępnej pamięci. Stosuje się tu przedstawione na rysunku 3.4 podejście podobne jak w algorytmie *sortowania przez scalanie*. Należy rozpocząć odczyt plików wejściowych, sprawdzić pierwszy klucz z każdego pliku, skopiować do pliku wyjściowego najmniejszy klucz (według porządku sortowania) i powtórzyć cały proces. W efekcie powstaje nowy scalony plik segmentu, który także jest posortowany według kluczy.
2. Co się dzieje, jeśli ten sam klucz znajduje się w kilku segmentach wejściowych? Pamiętaj, że każdy segment obejmuje wszystkie wartości zapisane w bazie w jakimś okresie. To oznacza, że wszystkie wartości z jednego segmentu wejściowego muszą być nowsze od wszystkich wartości z innego segmentu (przy założeniu, że zawsze scalane są przyległe segmenty). Gdy kilka segmentów obejmuje ten sam klucz, można zachować wartość z najnowszego segmentu i usunąć wartości ze starszych segmentów.
3. Aby znaleźć w pliku konkretny klucz, nie trzeba już przechowywać indeksu z wszystkimi kluczami w pamięci. Przykład pokazano na rysunku 3.5. Założmy, że szukasz klucza *handiwork*, ale nie znasz dokładnego przesunięcia tego klucza w pliku segmentu. Znasz jednak przesunięcia kluczy *handbag* i *handsome*, a ponieważ dane są posortowane, wiesz, że klucz *handiwork* musi występować między nimi. To oznacza, że możesz zacząć od przesunięcia dla klucza *handbag* i zacząć wyszukiwanie od tego miejsca do momentu znalezienia klucza *handiwork* (lub nieznalezienia, jeśli tego klucza nie ma w pliku).



4. Ponieważ żądania odczytu i tak wymagają przejrzenia kilku par klucz-wartość z żadanego przedziału, można pogrupować rekordy w bloki i skompresować je przed zapisem na dysku (na rysunku 3.5 blok wyróżniono ciemniejszym tłem). Każdy wpis w przechowywanym w pamięci indeksie rzadkim wskazuje wtedy początek skompresowanego bloku. Kompresja nie tylko zmniejsza ilość miejsca zajmowanego na dysku, ale też ogranicza liczbę operacji wejścia-wyjścia.

## Tworzenie i utrzymywanie plików SSTable

Na razie wszystko wydaje się łatwe. Jak jednak sprawić, by dane były posortowane według kluczy? Operacje zapisu mogą się odbywać w dowolnej kolejności.

Utrzymywanie posortowanej struktury na dysku jest możliwe (zob. punkt „B-drzewa”), jednak przechowywanie jej w pamięci jest znacznie łatwiejsze. Istnieje wiele znanych struktur drzewiastych, które można zastosować (np. drzewa czerwono-czarne lub drzewa AVL) [2]. Za pomocą tych struktur danych możesz wstawiać klucze w dowolnej kolejności i czytywać je według porządku sortowania.

Teraz można sprawić, by system składowania danych działał w następujący sposób:

- W trakcie zapisu danych należy dodać je do przechowywanego w pamięci drzewa zrównoważonego (np. drzewa czerwono-czarnego). Przechowywane w pamięci drzewo jest czasem nazywane *memtable*.
- Gdy struktura *memtable* staje się większa od pewnej wartości progowej (zwykle kilku megabajtów), należy ją zapisać na dysku jako plik SSTable. Można to zrobić w wydajny sposób, ponieważ drzewo obejmuje już pary klucz-wartość posortowane według klucza. Nowy plik SSTable staje się najnowszym segmentem bazy. W trakcie zapisywania tego pliku na dysku nowe operacje zapisu są przeprowadzane na nowej instancji struktury *memtable*.
- Aby obsłużyć żądanie odczytu, najpierw spróbuj znaleźć klucz w strukturze *memtable*, następnie w najnowszym segmencie na dysku, potem w kolejnym segmencie z dysku itd.
- Od czasu do czasu uruchamiaj w tle proces scalania i kompresji, aby łączyć pliki segmentów i usuwać zastąpione lub usunięte wartości.

Ten schemat sprawdza się bardzo dobrze. Występuje w nim tylko jeden problem: jeśli w bazie nastąpi awaria, najnowsze dane (znajdujące się w strukturze *memtable*, ale niezapisane na dysku) zostaną utracone. Aby uniknąć tego problemu, można przechowywać na dysku odrębny dziennik, w którym natychmiast umieszczane są wszystkie zapisywane dane (tak jak w rozwiązaniu z poprzedniego punktu). Ten dziennik nie jest posortowany, jednak nie ma to znaczenia, ponieważ jego jedyną funkcją to umożliwienie odzyskania struktury *memtable* po awarii. Za każdym razem, gdy struktura *memtable* jest zapisywana do pliku SSTable, powiązany dziennik można usuwać.

## Tworzenie drzew LSM na podstawie plików SSTable

Opisany tu algorytm jest używany w narzędziach LevelDB [6] i RocksDB [7]. Są to biblioteki do przechowywania par klucz-wartość zaprojektowane do stosowania w innych aplikacjach. LevelDB może być m.in. używana w systemie Riak zamiast narzędzia Bitcask. Podobne systemy składowania danych są używane w projektach Cassandra i HBase [8], które zostały zainspirowane tekstem na temat bazy Bigtable z firmy Google [9] (to w tym tekście wprowadzone zostały nazwy *SSTable* i *memtable*).

Jako pierwszy przedstawioną tu strukturę służącą do indeksowania opisał Patrick O’Neil ze współpracownikami. O’Neil użył nazwy *drzewo LSM* (ang. *Log-Structured Merge-Tree*) [10], bazując na wcześniejszej pracy na temat systemów plików o strukturze dziennika [11]. Systemy składowania danych wykorzystujące scalanie i kompresję posortowanych plików są czasem systemami LSM.

Lucene, system indeksujący na potrzeby wyszukiwania pełnotekstowego używany w narzędziach Elasticsearch i Solr, używa podobnej metody do przechowywania *słownika pojęć* [12, 13]. Indeks pełnotekstowy jest dużo bardziej złożony od indeksu z parami klucz-wartość, ale opiera się na tym samym pomysłe — na podstawie słowa z zapytania wyszukuje wszystkie dokumenty (strony internetowe, opisy produktów itd.), w których to słowo występuje. To rozwiązanie jest zaimplementowane za pomocą struktury z parami klucz-wartość, gdzie klucz to słowo (*pojęcie*), a wartość to lista identyfikatorów wszystkich zawierających je dokumentów (*lista wystąpień*). W systemie Lucene odwzorowanie z pojęć na listę wystąpień znajduje się w posortowanych plikach podobnych do plików SSTable. W razie potrzeby pliki te są scalane w tle [14].

### Optymalizacje wydajności

Jak zawsze wiele pracy wkłada się w zapewnienie wysokiej wydajności systemu składowania danych w praktyce. Na przykład algorytm dla drzew LSM może być wolny, gdy wyszukuje klucze nieistniejące w bazie. Trzeba wtedy sprawdzić strukturę memtable, a następnie wszystkie segmenty aż do najstarszego (co prawdopodobnie wymaga odczytu każdego z nich z dysku), aby się upewnić, że dany klucz nie istnieje. Aby zoptymalizować dostęp tego rodzaju, systemy składowania danych często używają dodatkowych *filtrów Blooma* [15]. Filtr Blooma to wydajna ze względu na pamięć struktura danych przybliżająca zawartość zbioru. Pozwala ona stwierdzić, że klucz nie występuje w bazie, i uniknąć tym samym wielu niepotrzebnych odczytów z dysku związanych z nieistniejącymi kluczami.

Istnieją też różne strategie określania kolejności i czasu kompresji oraz scalania plików SSTable. Najczęściej stosowane metody to kompresja *warstwami zależnymi od wielkości* (ang. *size-tiered*) i *poziomowa* (ang. *leveled*). W bibliotekach LevelDB i RocksDB używa się kompresji poziomej (stąd nazwa LevelDB), w HBase stosuje się kompresję warstwami zależnymi od wielkości, a Cassandra obsługuje obie te techniki [16]. W kompresji warstwami zależnymi od wielkości nowsze i mniejsze pliki SSTable są sukcesywnie scalane ze starszymi i większymi plikami. W kompresji poziomej klucze z całego zakresu są dzielone na grupy i umieszczane w mniejszych plikach SSTable, a starsze dane są przenoszone na odrębne „poziomy”, co pozwala na stopniową kompresję i zużycie mniejszej ilości pamięci.

Choć istnieje wiele subtelności, podstawowy pomysł związany z drzewami LSM — przechowywanie zestawu scalanych w tle plików SSTable — jest prosty i skuteczny. Rozwiązanie to działa dobrze nawet wtedy, gdy zbiór danych jest znacznie większy niż ilość dostępnej pamięci. Ponieważ dane są posortowane, można wydajnie wykonywać zapytania dotyczące zakresów (z przeszukiwaniem wszystkich kluczy od pewnej wartości minimalnej do określonej wartości maksymalnej), a ponieważ zapis na dysku odbywa się sekwencyjnie, drzewa LSM pozwala uzyskać bardzo wysoką wydajność zapisu.

## B-drzewa

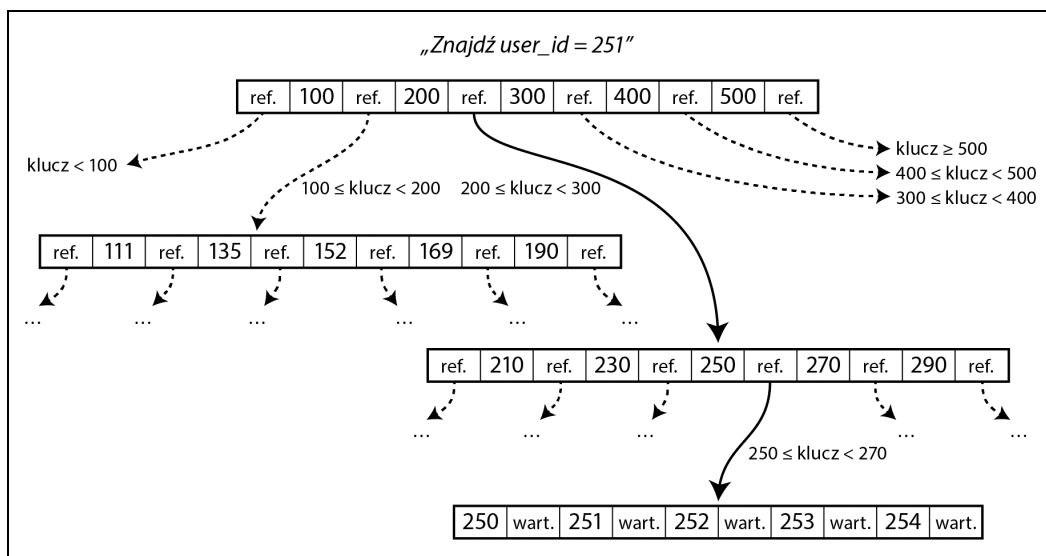
Opisywane do tej pory indeksy o strukturze dziennika zyskują popularność, jednak nie są najczęściej występującym rodzajem indeksu. W przypadku indeksów najczęściej stosuje się inną strukturę — *b-drzewo*.

Wprowadzone w 1970 r. [17] i nazwane „wszechobecnymi”, niecałe 10 lat później [18] b-drzewa bardzo dobrze poradziły sobie z testem czasu. Pozostają standardową implementacją indeksów w prawie wszystkich relacyjnych bazach danych, a także są stosowane w wielu bazach nierelacyjnych.

B-drzewa (podobnie jak pliki SSTable) przechowują pary klucz-wartość posortowane według kluczy, co umożliwia wydajne wyszukiwanie kluczy i wartości oraz obsługę zapytań opartych na przedziałach. Na tym jednak podobieństwa się kończą. W b-drzewach zastosowano zupełnie odmienną filozofię projektową.

Opisane wcześniej indeksy o strukturze dziennika dzielą bazę na *segmenty* o zmiennej wielkości (zwykle równej kilka megabajtów lub więcej) i zawsze zapisują segment sekwencyjnie. Natomiast w b-drzewach baza jest podzielona na *bloki* lub *strony* o stałej wielkości (tradycyjnie równej 4 KB, choć czasem większej), wczytywane lub zapisywane po jednej stronie naraz. Ten projekt jest bardziej zbliżony do używanego sprzętu, ponieważ dyski także są uporządkowane w bloki o stałej wielkości.

Każdą stronę można zidentyfikować za pomocą adresu lub lokalizacji, co umożliwia wskazywanie na jednej stronie innych stron. Przypomina to wskaźniki, ale używane na dysku, a nie w pamięci. Referencje do stron można wykorzystać do utworzenia drzewa stron, co pokazano na rysunku 3.6.



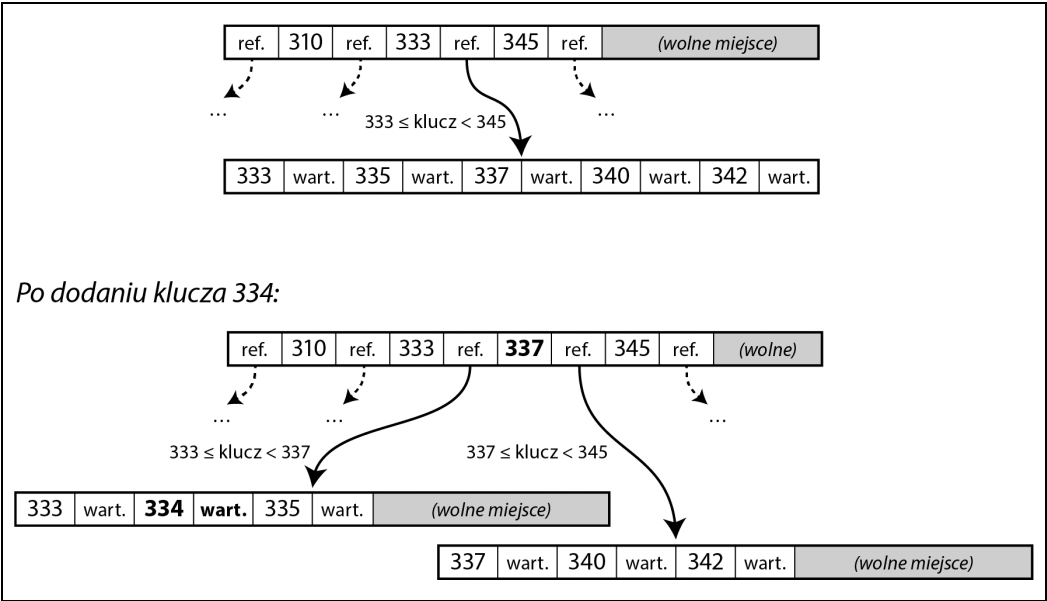
Rysunek 3.6. Wyszukiwanie klucza z użyciem indeksu opartego na b-drzewie

Jedna strona jest wyznaczona na *korzeń* b-drzewa. Zawsze gdy chcesz znaleźć klucz w indeksie, zaczynasz od korzenia. Ta strona obejmuje zbiór kluczy i referencje do stron podrzędnych. Każda taka strona odpowiada za klucze z określonego przedziału, a klucze między referencjami wyznaczają granice tych przedziałów.

W przykładzie z rysunku 3.6 szukany jest klucz 251, dlatego wiadomo, że trzeba wybrać referencję między granicami 200 a 300. To prowadzi do podobnej strony, która dalej dzieli przedział 200 – 300 na podprzedziały. Ostatecznie docieramy do strony zawierającej klucze (*strony-liścia*), która albo zawiera wartości powiązane z poszczególnymi kluczami, albo referencje do stron, gdzie można znaleźć te wartości.

Liczba referencji do stron podrzędnych na jednej stronie b-drzewa to *parametr rozgałęziania*. Na przykład na rysunku 3.6 parametr rozgałęziania wynosi sześć. W praktyce zależy on od ilości miejsca potrzebnego na referencje do stron i granice przedziałów. Zwykle wynosi kilkaset.

Jeśli chcesz zaktualizować wartość dla istniejącego klucza z b-drzewa, poszukaj strony-liścia zawierającej ten klucz, zmień wartość na tej stronie i ponownie zapisz stronę na dysku (wszystkie referencje do tej strony pozostają prawidłowe). Gdy chcesz dodać nowy klucz, musisz znaleźć stronę powiązaną z przedziałem obejmującym ten nowy klucz i dodać go do tej strony. Jeśli na stronie brakuje miejsca na dodanie nowego klucza, jest ona dzielona na dwie w połowie zapełnione strony, a strona nadrzędna jest aktualizowana, aby uwzględnić nowy podział przedziałów kluczy (zob. rysunek 3.7)<sup>2</sup>.



Rysunek 3.7. Powiększanie b-drzewa w wyniku podziału strony

Ten algorytm gwarantuje, że drzewo pozostaje *zrównoważone*. B-drzewo o  $n$  kluczach zawsze ma głębokość  $O(\log n)$ . Większość baz można zmieścić w b-drzewie mającym trzy lub cztery poziomy, dlatego nie trzeba używać wielu referencji, aby znaleźć szukaną stronę. Czteropoziomowe drzewo stron o wielkości 4 KB i parametrze rozgałęziania równym 500 pozwala zapisać do 256 TB danych.

<sup>2</sup> Wstawianie nowych kluczy do b-drzewa jest dość intuicyjne, jednak usuwanie ich (przy zachowaniu zrównoważonego charakteru drzewa) to bardziej złożona operacja [2].

## Zapewnianie niezawodności b-drzew

Podstawową operację zapisu w b-drzewach stanowi nadpisywanie strony na dysku nowymi danymi. Przyjmuje się, że taka operacja nie zmienia lokalizacji strony (oznacza to, że w procesie nadpisywania strony wszystkie referencje na niej pozostają nienaruszone). Pozostaje to w wyraźnym kontraście z indeksami o strukturze dziennika (takimi jak drzewa LSM), które jedynie dodają dane do plików (i ostatecznie usuwają nieaktualne pliki), ale nigdy nie modyfikują plików w miejscu.

Nadpisywanie strony na dysku możesz traktować jak operację sprzętową. Na magnetycznym dysku twardym polega ona na przeniesieniu głowicy w odpowiednie miejsce, odczekaniu na przesunięcie obrotowego talerza we właściwą pozycję i zastąpieniu odpowiedniego sektora nowymi danymi. Na dyskach SSD operacja jest bardziej skomplikowana, ponieważ wymaga wykasowania i ponownego zapisania stosunkowo dużych bloków układu z pamięcią [19].

Ponadto niektóre operacje wymagają zastąpienia kilku stron. Na przykład jeśli dzielisz stronę, ponieważ wstawianie danych spowodowało jej przepełnienie, musisz zapisać dwie strony (powstałe wskutek podziału), a także zastąpić stronę nadrzędną, aby zaktualizować referencje do dwóch stron podrzędnych. To niebezpieczna operacja — jeśli baza ulegnie awarii po zapisaniu tylko niektórych stron, otrzymasz uszkodzony indeks (może on np. obejmować *osieroconą* stronę, której nie odpowiada żadna strona nadrzędna).

Aby zapewnić odporność bazy na awarie, często w implementacjach b-drzew na dysku umieszczana jest dodatkowa struktura danych: *dziennik zapisu z wyprzedzeniem* (ang. *write-ahead log* — **WAL**; inna nazwa to *redo log*, czyli dziennik powtórzeń). To plik przeznaczony tylko do dodawania danych, w którym trzeba zapisać każdą zmianę w b-drzewie przed jej wprowadzeniem na stronach drzewa. Po przywróceniu bazy po awarii za pomocą tego dziennika można odzyskać spójny stan b-drzewa [5, 20].

Dodatkową komplikację związaną z aktualizowaniem stron w miejscu stanowi to, że konieczna jest staranna kontrola współbieżności, jeśli wiele wątków ma jednocześnie używać b-drzewa. W przeciwnym razie wątek może używać drzewa w niespójnym stanie. Na potrzeby kontroli współbieżności struktury danych drzewa zwykle są chronione za pomocą *zatrząsek* (ang. *latch*; są to proste blokady). Techniki ze strukturą dziennika są pod tym względem prostsze, ponieważ scalają dane w tle bez wpływu na przychodzące zapytania i od czasu do czasu atomowo zastępują starsze segmenty nowymi.

## Optymalizacje b-drzew

Ponieważ b-drzewa są dostępne od tak długiego czasu, nie zaskakuje fakt, że przez lata opracowano wiele optymalizacji. Oto kilka z nich:

- Zamiast nadpisywać strony i przechowywać dziennik zapisu z wyprzedzeniem na potrzeby odzyskiwania stanu po awarii, w niektórych bazach (np. w LMDB) używa się schematu kopiowania przy zapisie [21]. Zmodyfikowana strona jest zapisywana w innej lokalizacji i tworzy się nową wersję stron nadrzędnych z referencją prowadzącą do tej lokalizacji. To podejście przydaje się także do kontroli współbieżności, o czym się przekonasz w punkcie „Izolowanie snapshotów i powtarzalny odczyt”.

- Można zaoszczędzić miejsce na stronach, zapisując skróconą postać klucza. Zwłaszcza na wewnętrznych stronach drzewa klucze muszą zapewniać tylko tyle informacji, aby określać granice między przedziałami kluczy. Umieszczanie większej liczby kluczy na stronie pozwala zwiększyć parametr rozgałęziania, a tym samym zmniejszyć liczbę poziomów<sup>3</sup>.
- Ogólnie strony można umieścić na dysku w dowolnym miejscu. Nic nie wymaga, by strony ze zbliżonymi przedziałami kluczy znajdowały się na dysku blisko siebie. Jeśli zapytanie musi sprawdzić dużą część przedziału kluczy zgodnie z porządkiem sortowania, układ „strona obok strony” może być niewydajny, ponieważ każdą wczytywaną stronę trzeba znaleźć na dysku. Dlatego wiele implementacji b-drzew próbuje utworzyć układ drzewa w taki sposób, by strony-liście występowały na dysku sekwencyjnie. Jednak wraz z rozrastaniem się drzewa trudno zachować taki układ. Z kolei w drzewach LSM duże segmenty danych są ponownie zapisywane w jednej operacji w trakcie scalania, dlatego łatwiej wtedy utrzymać kolejne klucze blisko siebie na dysku.
- Do drzewa dołączone zostały dodatkowe wskaźniki. Na przykład każda strona-liść może obejmować referencje do lewej i prawej strony z tego samego poziomu, co umożliwia przeglądanie kluczy po kolei bez wracania do stron nadrzędnych.
- W odmianach b-drzew takich jak *drzewa fraktalne* [22] zapożyczono pewne pomysły z obszaru dzienników, aby ograniczyć liczbę operacji wyszukiwania na dysku (drzewa te nie mają nic wspólnego z fraktalami).

## Porównanie b-drzew z drzewami LSM

Choć implementacje b-drzew są zwykle bardziej dojrzałe niż implementacje drzew LSM, te ostatnie też są ciekawe z powodu charakterystyki działania. Ogólnie uważa się, że drzewa LSM umożliwiają szybszy zapis, a b-drzewa pozwalają na szybszy odczyt [23]. Odczyt w drzewach LSM jest zwykle wolniejszy, ponieważ na różnych etapach kompresji trzeba sprawdzić kilka różnych struktur danych i plików SSTable.

Jednak testy porównawcze często okazują się niejednoznaczne i podatne na szczegółową charakterystykę obciążenia roboczego. Musisz testować systemy z użyciem konkretnego obciążenia roboczego, aby móc dokonać wiarygodnych porównań. W tym punkcie pokrótce omówiono kilka kwestii, które warto uwzględnić w trakcie pomiaru wydajności systemu składowania danych.

### Zalety drzew LSM

W indeksie używającym b-drzewa każdą porcję danych trzeba zapisać przynajmniej dwukrotnie: raz w dzienniku zapisu z wyprzedzeniem i raz na stronie drzewa (i czasem ponownie w momencie podziału stron). Występują też koszty związane z koniecznością zapisu całej strony nawet wtedy, gdy na stronie zmieniło się tylko kilka bajtów. Niektóre systemy składowania danych nadpisują tę samą stronę dwukrotnie, aby uniknąć powstania częściowo zaktualizowanej strony po awarii zasilania [24, 25].

---

<sup>3</sup> Ta odmiana bywa nazywana drzewami  $B^+$ , choć ta optymalizacja jest tak popularna, że tej wersji często nie odróżnia się od innych rodzajów b-drzew.



Indeksy o strukturze dziennika także wielokrotnie zapisują dane, co wynika z powtarzania kompresji i scalania plików SSTable. Ten efekt (jeden zapis w bazie powodujący wiele zapisów na dysku w trakcie użytkowania tej bazy) jest nazywany mnożnikiem zapisu (ang. *write amplification*). Stwarza on problemy zwłaszcza w dyskach SSD, które mogą nadpisywać bloki tylko ograniczoną liczbę razy przed zużyciem.

W aplikacjach z dużą liczbą zapisów wąskim gardłem może być maksymalna szybkość zapisu przez bazę danych na dysku. Wtedy mnożnik zapisu ma bezpośredni wpływ na wydajność. Im więcej informacji system składowania danych zapisuje na dysku, tym mniej zapisów na sekundę potrafi obsłużyć przy określonej szybkości dysku.

Ponadto drzewa LSM zwykle potrafią utrzymać wyższą przepustowość zapisu niż b-drzewa — po części dlatego, że czasem mają niższy mnożnik zapisu (choć zależy to od konfiguracji i obciążenia roboczego systemu składowania danych), a po części dlatego, że sekwencyjnie zapisują skompresowane pliki SSTable i nie muszą nadpisywać kilku stron drzewa [26]. Ta różnica jest ważna zwłaszcza w magnetycznych dyskach twardych, gdzie zapis sekwencyjny odbywa się znacznie szybciej niż losowy.

Drzewa LSM można bardziej skompresować, tworząc w ten sposób na dysku mniejsze pliki niż przy użyciu b-drzew. Systemy składowania danych używające b-drzew część dysku pozostawiają niewykorzystaną, co jest spowodowane fragmentacją. Gdy strona jest dzielona lub gdy nie można zmieścić wiersza na istniejącej stronie, część przestrzeni na stronie pozostaje niewykorzystana. Ponieważ drzewa LSM nie są oparte na stronach i okresowo zastępują pliki SSTable w celu ograniczenia fragmentacji, powodują niższe koszty związane z przestrzenią dyskową — zwłaszcza wtedy, gdy stosuje się kompresję poziomową [27].

W wielu dyskach SSD oprogramowanie sprzętowe wewnętrznie używa algorytmu opartego na strukturze dziennika, aby przekształcać zapisy losowe w zapisy sekwencyjne, dlatego wpływ wzorca zapisów w systemie składowania danych jest tu mniej odczuwalny [19]. Jednak niższy mnożnik zapisu i ograniczona fragmentacja są zaletami także w dyskach SSD. Bardziej zwarte reprezentowanie danych umożliwia obsługę większej liczby żądań odczytu i zapisu przy danej przepustowości operacji wejścia-wyjścia.

### Wady drzew LSM

Wadę przechowywania z użyciem struktury dziennika stanowi to, że proces kompresji może obniżać szybkość wykonywanych odczytów i zapisów. Choć systemy składowania danych próbują kompresować dane przyrostowo i bez wpływu na dostęp współbieżny, dyski mają ograniczone możliwości, dlatego może się zdarzyć, że żądanie będzie musiało czekać na zakończenie przez dysk kosztownej operacji kompresji. Wpływ na przepustowość i średni czas odpowiedzi jest zwykle niski, jednak dla wyższych percentyli (zob. punkt „Opis wydajności”) czas odpowiedzi dla zapytań w systemach przechowywania używających struktury dziennika bywa wysoki. B-drzewa mogą się okazać bardziej przewidywalne [28].

Inny problem z kompresją występuje przy wysokiej przepustowości zapisu. Ograniczoną przepustowość zapisu na dysku trzeba dzielić między początkowy zapis (zapis w dzienniku i przenoszenie struktury memtable na dysk) a działające w tle wątki kompresujące. W trakcie zapisu danych w pustej bazie dla początkowego zapisu można wykorzystać całą przepustowość dysku. Jednak im baza staje się większa, tym więcej przepustowości trzeba przeznaczać na kompresję.

Jeśli przepustowość zapisu jest wysoka, a kompresja nie została starannie skonfigurowana, może się zdarzyć, że kompresja nie nadąży za szybkością nowych zapisów. Wtedy liczba niescalonych segmentów na dysku rośnie do czasu wyczerpania się miejsca. Ponadto skutkuje to spowolnieniem odczytów, ponieważ trzeba sprawdzać większą liczbę plików segmentów. Systemy oparte na plikach SSTable zwykle nie ograniczają szybkości nowych zapisów nawet wtedy, gdy kompresja nie potrafi za nimi nadążyć. Dlatego trzeba bezpośrednio monitorować system w celu wykrywania takich sytuacji [29, 30].

Zaletę b-drzew stanowi to, że każdy klucz występuje w indeksie tylko w jednym miejscu. W systemie składowania danych używającym struktury dziennika ten sam klucz może się znajdować w wielu różnych segmentach. To sprawia, że b-drzewa są atrakcyjne w bazach, które mają zapewniać semantykę transakcji. W wielu bazach relacyjnych izolacja transakcji jest implementowana z użyciem blokad dotyczących przedziałów kluczy. W indeksie opartym na b-drzewie te blokady można bezpośrednio powiązać z drzewem [5]. W rozdziale 7. znajdziesz szczegółowe omówienie tego zagadnienia.

B-drzewa są ważnym elementem architektury baz danych i zapewniają stabilnie wysoką wydajność dla wielu rodzajów obciążenia roboczego. Dlatego jest mało prawdopodobne, że w bliskiej przyszłości przestaną być używane. W nowych magazynach danych coraz popularniejsze stają się indeksy o strukturze dziennika. Nie istnieje szybki i łatwy sposób na ustalenie, którego rodzaju system składowania danych będzie lepszy na potrzeby Twoich zastosowań, dlatego warto przeprowadzić testy empiryczne.

## Inne struktury używane w przypadku indeksów

Do tej pory opisywane były tylko indeksy z kluczami i wartościami, które przypominają oparty na *kluczu głównym* indeks z modelu relacyjnego. Klucz główny to unikatowy identyfikator wiersza w tabeli relacyjnej, jednego dokumentu w bazie opartej na dokumentach lub jednego wierzchołka w bazie opartej na grafach. Inne rekordy w bazie mogą wskazywać ten wiersz, dokument lub wierzchołek za pomocą klucza głównego (identyfikatora), a indeks służy do interpretowania takich referencji.

Bardzo często występują też *indeksy pomocnicze*. W bazach relacyjnych możesz utworzyć kilka indeksów pomocniczych dla tej samej tabeli, używając polecenia `CREATE INDEX`. Takie indeksy są często niezbędne do wydajnego przeprowadzania złączeń. Na przykład w bazie z rysunku 2.1 z rozdziału 2. zapewne używany byłby indeks pomocniczy w przypadku kolumn `user_id`, co pozwala znaleźć w różnych tabelach wszystkie wiersze dotyczące tego samego użytkownika.

Indeks pomocniczy można łatwo zbudować na podstawie indeksu z kluczami i wartościami. Główna różnica polega na tym, że w indeksie pomocniczym klucze nie są unikatowe. Oznacza to, że może występować wiele wierszy (dokumentów, wierzchołków) o tym samym kluczu. Istnieją tu dwa rozwiązania: można albo każdej wartości z indeksu przypisać listę pasujących identyfikatorów wierszy (przypomina to listę wystąpień z indeksu pełnotekstowego), albo utworzyć unikatowe klucze, dodając do nich identyfikatory wierszy. W obu podejściach jako pomocnicze indeksy można zastosować zarówno b-drzewa, jak i indeksy o strukturze dziennika.

## Przechowywanie wartości w indeksie

Klucz w indeksie to element szukany w zapytaniu. Wartością mogą być dwie rzeczy: sam wiersz (dokument lub wierzchołek) lub referencja do wiersza zapisanego w innym miejscu. W tym drugim przypadku miejscem przechowywania wierszy jest *plik nieuporządkowany*. Przechowuje on dane w nieustalonej kolejności (może to być plik wyłącznie z możliwością dodawania danych lub plik śledzący usuwane wiersze, aby móc je później zastąpić nowymi danymi). Podejście z plikiem nieuporządkowanym często się stosuje, ponieważ pozwala uniknąć duplikowania danych, gdy istnieje wiele indeksów pomocniczych. Każdy indeks wskazuje wtedy miejsce w pliku nieuporządkowanym, a same dane są przechowywane w jednym miejscu.

Gdy aktualizujesz wartość bez modyfikowania klucza, podejście z plikiem nieuporządkowanym może się okazać wydajne. Rekord można nadpisać w miejscu — pod warunkiem że nowa wartość nie jest dłuższa od dawnej. Sytuacja się komplikuje, jeśli nowa wartość jest dłuższa, ponieważ zwykle trzeba ją wtedy przenieść w nowe miejsce pliku, gdzie dostępna jest wystarczająca ilość miejsca. Wtedy trzeba albo zaktualizować wszystkie indeksy, aby prowadziły do nowej lokalizacji rekordu w pliku nieuporządkowanym, albo umieścić w starej lokalizacji wskaźnik pośredni [5].

W niektórych sytuacjach dodatkowe przejście od indeksu do pliku nieuporządkowanego oznacza zbyt duży spadek wydajności odczytu. Dlatego pożądane może być przechowywanie indeksowanych wierszy bezpośrednio w indeksie. Służy do tego *indeks klastrowy*. Na przykład w systemie InnoDB z MySQL-a dla klucza głównego tabeli zawsze tworzony jest indeks klastrowy, a indeksy pomocnicze używają tego klucza głównego (zamiast lokalizacji z pliku nieuporządkowanego) [31]. W systemie SQL Server można tworzyć jeden indeks klastrowy na tabelę [32].

Kompromisem między indeksem klastrowym (przechowującym w indeksie wszystkie dane z wierszy) a nieklastrowym (przechowującym w indeksie tylko referencje do danych) jest *indeks pokrywający* (lub *indeks z dołączonymi kolumnami*), w którym przechowywane są *niektóre* kolumny tabeli [33]. Dzięki temu niektóre zapytania można obsługiwać za pomocą samego indeksu (w takiej sytuacji indeks *pokrywa* zapytanie) [32].

Indeksy klastrowe i pokrywające (podobnie jak inne techniki z powielaniem danych) mogą przyspieszać odczyty, jednak wymagają dodatkowego miejsca i mogą zwiększać koszty zapisu. Bazy muszą też wykonywać dodatkowe operacje w celu wymuszania gwarancji związanych z transakcjami, ponieważ aplikacje nie powinny widzieć niespójności spowodowanych duplikatami.

## Indeksy wielokolumnowe

Indeksy omawiane do tego miejsca odwzorowują tylko jeden klucz na wartość. To nie wystarcza, gdy zapytanie dotyczy jednocześnie wielu kolumn tabeli (lub wielu pól dokumentu).

Najczęściej spotykanym rodzajem indeksu wielokolumnowego jest indeks połączony (ang. *concatenated index*), który łączy kilka pól w jeden klucz, scalając kolumny ze sobą (definicja indeksu określa kolejność scalanych pól). Przypomina to staroświecką książkę telefoniczną, która zapewnia indeks w postaci (*nazwisko, imię*) dla numerów telefonów. Z powodu porządku sortowania ten indeks można wykorzystać do znalezienia wszystkich osób o określonym nazwisku lub z daną kombinacją *nazwisko-imię*. Jednak ten indeks jest bezużyteczny, jeśli chcesz znaleźć wszystkie osoby o określonym imieniu.

Indeksy wielowymiarowe to ogólniejszy sposób obsługi zapytań dotyczących kilku kolumn. Technika ta jest ważna zwłaszcza w kontekście danych geoprzestrzennych. Na przykład witryna do wyszukiwania restauracji może używać bazy z szerokością i długością geograficzną każdego lokalu. Gdy użytkownik szuka restauracji na mapie, witryna musi znaleźć wszystkie restauracje w prostokątnym obszarze, który użytkownik wyświetlił. Wymaga to dwuwymiarowego zapytania z wykorzystaniem przedziałów:

```
SELECT * FROM restaurants WHERE latitude > 51.4946 AND latitude < 51.5079  
AND longitude > -0.1162 AND longitude < -0.1004;
```

Standardowy indeks oparty na b-drzewie lub drzewie LSM nie umożliwia wydajnego obsłużenia takiego zapytania. Taki indeks może zwrócić albo wszystkie restauracje dla określonego przedziału szerokości (ale dla dowolnej długości), albo wszystkie restauracje dla danego przedziału długości (ale gdziekolwiek między biegunami północnym a południowym), ale nie pozwala uwzględnić obu wymiarów jednocześnie.

Jedna możliwość to przekształcanie dwuwymiarowej lokalizacji na jedną wartość (z użyciem krzywej wypełniającej) i zastosowanie standardowego indeksu z b-drzewem [34]. Jednak częściej używane są wyspecjalizowane indeksy przestrzenne takie jak r-drzewa. Na przykład w rozszerzeniu PostGIS indeksy geoprzestrzenne są zaimplementowane jako r-drzewa z użyciem indeksów **GiST** (ang. *Generalized Search Tree*) z bazy PostgreSQL [35]. Nie ma tu miejsca na szczegółowe omawianie r-drzew, dostępna jest jednak obszerna literatura na ich temat.

Ciekawe że indeksy wielowymiarowe są przeznaczone nie tylko do lokalizacji geograficznych. Na przykład w witrynie sklepu internetowego można używać trójwymiarowego indeksu dla wymiarów (*czerwony, zielony, niebieski*) do wyszukiwania produktów o kolorach z określonego przedziału. Z kolei w bazie obserwacji meteorologicznych można utworzyć dwuwymiarowy indeks (*data, temperatura*), aby móc wydajnie znaleźć np. wszystkie obserwacje z 2013 r., w których temperatura wynosiła od 25 do 30°C. Gdy używa się indeksu jednowymiarowego, trzeba albo pobierać wszystkie rekordy z 2013 r. (niezależnie od temperatury), a następnie filtrować je na podstawie temperatury, albo na odwrót. Indeks dwuwymiarowy pozwala zawęzić zbiór danych jednocześnie na podstawie znacznika czasu i temperatury. Ta technika jest stosowana w bazie HyperDex [36].

## Wyszukiwanie pełnotekstowe i indeksy rozmyte

We wszystkich omawianych do tego miejsca indeksach przyjmowano, że dostępne są dokładne dane umożliwiające zapytania o precyzyjne wartości klucza lub przedział wartości klucza przy określonym porządku sortowania. Takie indeksy nie umożliwiają jednak wyszukiwania *podobnych* kluczy, np. błędnie zapisanych słów. Tego rodzaju *rozmyte* zapytania wymagają innych technik.

Na przykład wyszukiwarki pełnotekstowe często umożliwiają rozbudowanie wyszukiwania jednego słowa o jego synonimy, ignorowanie wersji słowa uzyskanych za pomocą odmiany, wyszukiwanie wystąpień słów znajdujących się blisko siebie w tym samym dokumencie, a także obsługują różne inne funkcje zależne od analiz lingwistycznych tekstu. Aby radzić sobie z literówkami w dokumentach lub zapytaniach, Lucene umożliwia wyszukiwanie w tekście słów o określonej odległości edycyjnej (odległość edycyjna równa 1 oznacza, że dodano, usunięto lub zastąpiono jedną literę) [37].

W punkcie „Tworzenie drzew LSM na podstawie plików SSTable” wspomniano, że w Lucene dla słownika pojęć używa się struktury podobnej do plików SSTable. Ta struktura wymaga niewielkiego przechowywanego w pamięci indeksu, który informuje zapytania o tym, jakie jest przesunięcie klucza w posortowanym pliku. W bazie LevelDB tym przechowywanym w pamięci indeksem jest rzadka kolekcja niektórych kluczy, jednak w Lucene ten indeks to automat skończenie stanowy, który przetwarza znaki z kluczy i jest podobny do *drzewa trie* [38]. Ten automat można przekształcić w *automat Levenshteina* służący do wydajnego wyszukiwania słów o określonej odległości edycyjnej [39].

Inne techniki wyszukiwania rozmytego są rozwijane w kierunku klasyfikacji dokumentów i uczenia maszynowego. Szczegóły znajdziesz w podręczniku poświęconym pobieraniu informacji [np. 40].

## Utrzymywanie wszystkiego w pamięci

Struktury danych opisane do tego miejsca stanowią odpowiedź na ograniczenia dysków. W porównaniu z główną pamięcią obsługa dysków jest kłopotliwa. Zarówno dyski magnetyczne, jak i dyski SSD wymagają starannego rozmieszczenia danych, jeśli chcesz uzyskać wysoką wydajność odczytu i zapisu. Ta niedogodność jest jednak tolerowana, ponieważ dyski mają dwie ważne zalety: są trwałe (ich zawartość nie zostaje utracona po wyłączeniu zasilania) i tańsze w przeliczeniu na gigabajt niż pamięć RAM.

Ponieważ pamięć RAM coraz bardziej tanieje, argument ceny za gigabajt traci na znaczeniu. Wiele zbiorów danych ma umiarkowaną wielkość, dlatego można je w całości przechowywać w pamięci (potencjalnie rozdzielonych między kilka maszyn). To doprowadziło do powstania *baz działających w pamięci*.

Niektóre magazyny par klucz-wartość działające w pamięci (np. Memcached) są przeznaczone wyłącznie na pamięć podręczną. Akceptowalna jest wtedy utrata danych po wznowieniu pracy przez maszynę. Jednak inne przechowywane w pamięci bazy mają zapewniać trwałość, co można osiągnąć za pomocą specjalnego sprzętu (np. pamięci RAM z zasilaniem baterijnym), zapisując na dysku dziennik zmian, okresowo zapisując na dysku snapshoty lub replikując przechowywany w pamięci stan w innych maszynach.

Gdy przechowywana w pamięci baza jest ponownie uruchamiana, musi wczytać stan — albo z dysku, albo przez sieć z repliki (chyba że używany jest specjalny sprzęt). Mimo zapisu danych na dysku to baza przechowywana w pamięci, ponieważ dysk jest używany tylko do zapewniania trwałości jako dziennik wyłącznie z możliwością dodawania danych, a odczyty są obsługiwane wyłącznie za pomocą pamięci. Zapis na dysku ma też zalety operacyjne: pliki na dysku można łatwo archiwizować, badać i analizować za pomocą zewnętrznych narzędzi.

Produkty takie jak VoltDB, MemSQL i Oracle TimesTen to przechowywane w pamięci bazy o modelu relacyjnym. Ich producenci twierdzą, że bazy te zapewniają znaczną poprawę wydajności, ponieważ eliminują wszystkie koszty związane z zarządzaniem strukturami danych przechowywanymi na dysku [41, 42]. RAMCloud to otwarty, przechowywany w pamięci magazyn kluczy i wartości z funkcją utrwalania (używający struktury dziennika zarówno dla danych w pamięci, jak i dla danych na dysku) [43]. Redis i Couchbase zapewniają słabe gwarancje trwałości, ponieważ asynchronicznie zapisują dane na dysku.

Niezgodnie z intuicją poprawa wydajności oferowana przez przechowywane w pamięci bazy nie wynika z tego, że nie muszą one wczytywać danych z dysku. Jeśli dostępna jest wystarczająca ilość pamięci, to nawet systemy składowania danych używające dysku czasem nie muszą wczytywać z niego informacji, ponieważ system operacyjny i tak zapisuje niedawno używane bloki dysku w pamięci podręcznej. Większa wydajność wynika z tego, że można uniknąć kosztów kodowania struktur danych z pamięci do postaci możliwej do zapisania na dysku [44].

Obok wydajności innym ciekawym aspektem baz danych przechowywanych w pamięci jest zapewnianie modeli danych, które trudno zaimplementować za pomocą indeksów opartych na dysku. Na przykład Redis udostępnia przypominający bazy interfejs do obsługi różnych struktur danych takich jak kolejki priorytetowe i zbiory. Ponieważ wszystkie dane są przechowywane w pamięci, implementacja tego interfejsu jest stosunkowo prosta.

Z niedawnych badań wynika, że architekturę baz przechowywanych w pamięci można rozbudować o obsługę zbiorów danych większych niż dostępna pamięć bez wywoływania kosztów związanych z architekturą opartą na dysku [45]. Metoda *rezygnacji z pamięci podręcznej* (ang. *anti-caching*) działa w taki sposób, że gdy brakuje pamięci, najdawniej używane dane są przenoszone z niej na dysk i wczytywane przy dostępie do nich w przyszłości. Podobnie działają pamięć wirtualna i pliki wymiany w systemie operacyjnym, jednak baza potrafi zarządzać pamięcią wydajniej niż system operacyjny, ponieważ może działać na poziomie poszczególnych rekordów, a nie całych stron pamięci. To podejście wymaga jednak, aby w pamięci mieściły się całe indeksy (tak jak w przykładzie z użyciem narzędzia Bitcask z początku rozdziału).

Jeśli technologie **NVM** (ang. *non-volatile memory*) zyskają na popularności, prawdopodobnie potrzebne będą dalsze zmiany w projekcie systemu składowania danych [46]. Obecnie jest to nowy obszar badań, jednak warto go obserwować w przyszłości.

## Przetwarzanie transakcji czy analityka?

W początkowym okresie przetwarzania danych biznesowych zapis w bazie zwykle odpowiadał *transakcji komercyjnej*: sprzedaży, złożeniu zamówienia u dostawcy, wypłacie wynagrodzenia pracownikowi itd. Nawet gdy bazy zaczęły być używane w obszarach, które nie wymagały przekazywania pieniędzy, pojęcie *transakcja* nadal pozostało w użyciu, oznaczając grupę odczytów i zapisów tworzących jednostkę logiczną.



Transakcja nie musi mieć właściwości **ACID** (ang. *atomicity, consistency, isolation i durability*, czyli atomowość, spójność, izolacja i trwałość). *Przetwarzanie transakcji* oznacza jedynie umożliwianie klientom odczytów i zapisów z małym opóźnieniem w odróżnieniu od zadań *przetwarzanych wsadowo*, uruchamianych tylko okresowo (np. raz dziennie). Właściwości ACID są opisane w rozdziale 7., a przetwarzanie wsadowe w rozdziale 10.

Choć bazy zaczęły być używane dla wielu rodzajów danych (komentarze do artykułów z blogów, działań w grach, kontaktów w książce adresowej itd.), podstawowy wzorzec dostępu pozostał podobny jak przy przetwarzaniu transakcji biznesowych. Aplikacja zwykle wyszukuje niewielką liczbę rekordów na podstawie klucza z użyciem indeksu. Rekordy są wstawiane lub aktualizowane na podstawie danych wejściowych od użytkownika. Ponieważ aplikacje są interaktywne, ten wzorzec dostępu jest nazywany **OLTP** (ang. *online transaction processing*, czyli przetwarzanie transakcji w czasie rzeczywistym).

Jednak bazy coraz częściej używane są też do *analitiky danych*, gdzie wzorce dostępu wyglądają zupełnie inaczej. Zapytanie analityczne zwykle wymaga przejrzenia bardzo dużej liczby rekordów, wczytania niewielkiej liczby kolumn z rekordów i obliczenia zagregowanych statystyk (np. liczby wystąpień, sumy lub średniej) zamiast zwracania użytkownikowi surowych danych. Na przykład jeśli dane znajdują się w tabeli z transakcjami sprzedaży, zapytania analityczne mogą wyglądać tak:

- Ile wyniósł łączny przychód w każdym z naszych sklepów w styczniu?
- O ile więcej niż zwykle sprzedaliśmy bananów w trakcie najnowszej promocji?
- Jaka marka posiłków dla niemowląt jest najczęściej kupowana razem z pieluchami marki X?

Takie zapytania są często pisane przez analityków biznesowych i uwzględniane w raportach, które pomagają zarządowi firmy podejmować lepsze decyzje (*analitika biznesowa*). Aby odróżnić ten wzorec używania baz od przetwarzania transakcji, nazwano go **OLAP** (ang. *online analytic processing*, czyli przetwarzanie analityczne w czasie rzeczywistym) [47]<sup>4</sup>. Różnice między podejściami OLTP i OLAP nie zawsze są wyraźne, jednak typowe cechy obu modeli wymieniono w tabeli 3.1.

Tabela 3.1. Porównanie cech przetwarzania transakcji i systemów analitycznych

Cecha	Systemy OLTP	Systemy OLAP
Podstawowy wzorec odczytu	Niewielka liczba rekordów na zapytanie; pobieranie na podstawie klucza	Agregowanie z użyciem dużej liczby rekordów
Podstawowy wzorec zapisu	Dostęp losowy, zapis z niskim opóźnieniem na podstawie danych wejściowych od użytkownika	Import masowy (ETL) lub strumieniowanie zdarzeń
Używany głównie przez	Użytkowników końcowych (klientów) z poziomu aplikacji sieciowej	Wewnętrznych analityków w celu wspomagania podejmowania decyzji
Dane reprezentują	Najnowszy stan danych (aktualny moment w czasie)	Historię zdarzeń zachodzących w czasie
Wielkość zbioru danych	Od gigabajtów do terabajtów	Od terabajtów do petabajtów

Początkowo dla zapytań OLTP i OLAP używane były te same bazy. SQL okazał się całkiem elastyczny w tym zakresie — działa dobrze zarówno dla zapytań OLTP, jak i dla zapytań OLAP. Mimo to pod koniec lat 80. i na początku lat 90. firmy zaczęły rezygnować z używania systemów OLTP na potrzeby analitiky na rzecz stosowania do analiz odrębnych baz. Te odrębne bazy zostały nazwane *hurtowniami danych*.

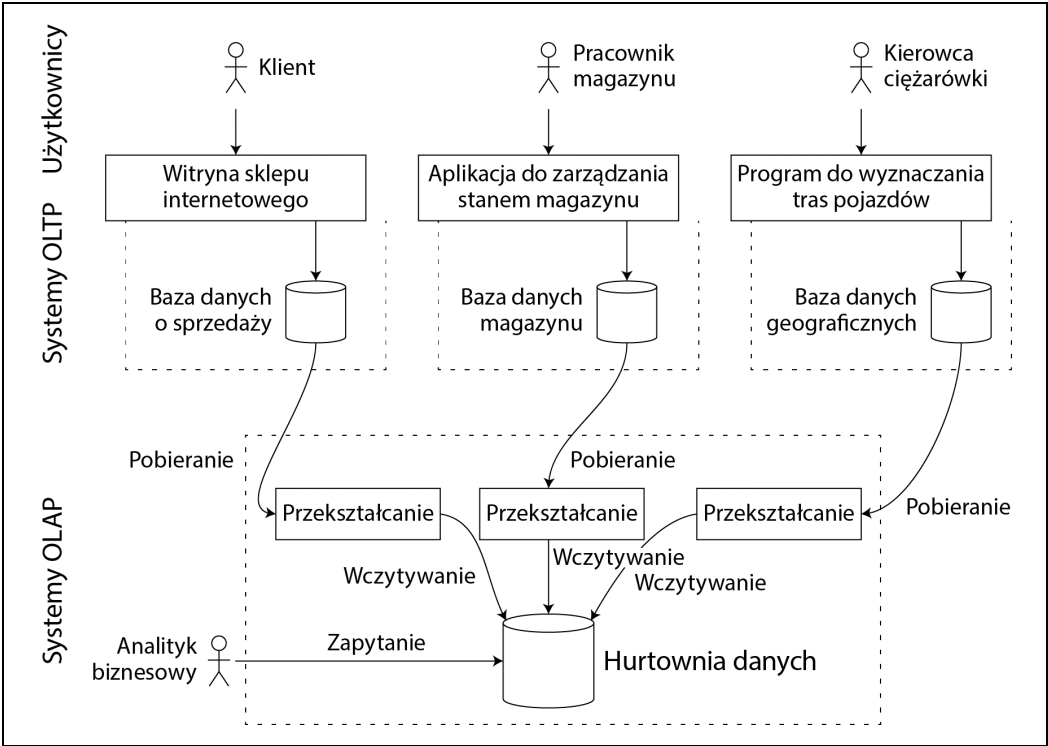
## Hurtownie danych

Firma może używać dziesiątek różnych systemów przetwarzania transakcji: systemów obsługujących udostępnianą użytkownikom witrynę, kontrolujących systemy w punktach sprzedaży (kasy) w fizycznych sklepach, śledzących stan magazynu, planujących trasy pojazdów, zarządzających dostawami i pracownikami itd. Każdy z tych systemów jest złożony i wymaga zespołu osób do konserwacji. Dlatego takie systemy zwykle działają niezależnie od siebie.

<sup>4</sup> Znaczenie słowa *online* w nazwie OLAP jest niejasne. Prawdopodobnie chodzi o to, że zapytania są przeznaczone nie tylko dla wcześniej zdefiniowanych raportów. Zamiast tego analitycy używają systemu OLAP interaktywnie na potrzeby zapytań eksploracyjnych.

Od systemów OLTP zwykle oczekuje się, że będą wysoce dostępne i będą przetwarzały transakcje z niewielkim opóźnieniem, ponieważ często są bardziej kluczowe dla działania firmy. Dlatego administratorzy baz ściśle chronią swoje bazy OLTP. Zazwyczaj niechętnie pozwalają analitykom biznesowym na uruchamianie doraźnych zapytań analitycznych w bazach OLTP, ponieważ takie zapytania są często kosztowne i skanują duże części zbioru danych, co może zmniejszyć wydajność jednocześnie przeprowadzanych transakcji.

Natomiast *hurtownia danych* to odrębna baza, do której analitycy mogą swobodnie kierować zapytania, nie wpływając na działanie baz OLTP [48]. Hurtownia danych obejmuje przeznaczoną tylko do odczytu kopię danych z różnych systemów OLTP firmy. Dane są pobierane z baz OLTP (albo za pomocą okresowego zrzutu danych, albo w ramach ciągłego strumienia aktualizacji), przekształcane na format ułatwiający analizy, oczyszczane, a następnie wczytywane do hurtowni danych. Proces przenoszenia danych do hurtowni to ETL (ang. *Extract-Transform-Load*, czyli pobieranie, przekształcanie, wczytywanie). Przedstawia go rysunek 3.8.



Rysunek 3.8. Uproszczony zarys procesu ETL wykonywanego na potrzeby hurtowni danych

Obecnie hurtownie danych są używane w prawie wszystkich dużych firmach, jednak w małych organizacjach stanowią rzadkość. Prawdopodobnie wynika to z tego, że większość małych firm nie używa tak wielu różnych systemów OLTP i ma niewielką ilość danych — na tyle małą, że możliwa jest obsługa zapytań za pomocą konwencjonalnej bazy SQL-owej lub nawet analiza danych w arkuszu kalkulacyjnym. W dużych firmach trzeba wykonać wiele skomplikowanych operacji, aby zrobić coś, co w niewielkiej instrukcji jest proste.



Dużą zaletą używania odrębnej hurtowni danych (w porównaniu z bezpośrednim kierowaniem przez analityków zapytań do systemów OLTP) jest to, że hurtownię można zoptymalizować pod kątem wzorców dostępu używanych w analityce. Okazuje się, że algorytmy indeksowania omówione w pierwszej połowie tego rozdziału dobrze sprawdzają się w systemach OLTP, ale nie są równie skuteczne w kontekście odpowiadania na zapytania analityczne.

Resztę rozdziału poświęcono na omówienie systemów składowania danych zoptymalizowanych na potrzeby analityki.

### Podział na bazy OLTP i hurtownie danych

W hurtowniach danych używa się najczęściej modelu relacyjnego, ponieważ SQL zwykle dobrze nadaje się do tworzenia zapytań analitycznych. Istnieje wiele narzędzi do wizualnej analizy danych, które generują zapytania w SQL-u, wizualizują wyniki i umożliwiają analitykom eksplorację danych (za pomocą operacji takich jak *drążenie danych* — ang. *drill-down* i *analiza według różnych przekrojów* — ang. *slicing and dicing*).

Hurtownia danych i relacyjna baza OLTP na pozór wyglądają podobnie, ponieważ w obu interfejsem do tworzenia zapytań jest SQL. Jednak wewnętrzne mechanizmy tych systemów mogą się wyraźnie różnić, ponieważ są zoptymalizowane pod kątem zupełnie odmiennych wzorców zapytań. Wiele producentów baz danych koncentruje się obecnie na obsłudze albo przetwarzania transakcji, albo obciążenia roboczego związanego z analityką, ale nie obu tych obszarów.

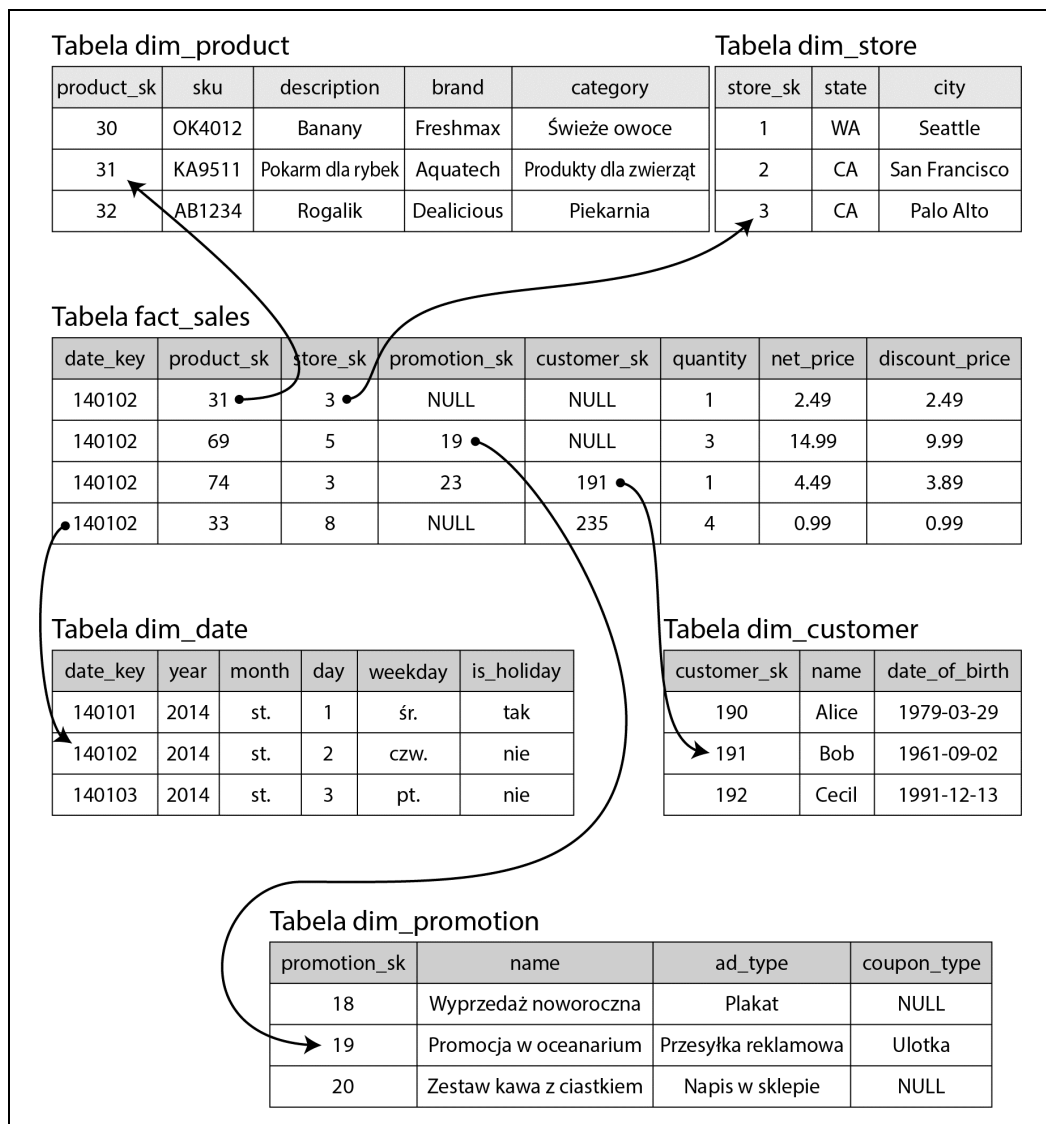
Niektóre bazy, np. Microsoft SQL Server i SAP HANA, oferują przetwarzanie transakcji i możliwości hurtowni danych w jednym produkcie. Jednak coraz częściej używane są do tego dwa odrębne systemy składowania danych i obsługi zapytań, dostępne za pomocą wspólnego interfejsu SQL-owego [49, 50, 51].

Producenci hurtowni danych (tacy jak Teradata, Vertica, SAP HANA i ParAccel) zwykle sprzedają swoje systemy z drogimi licencjami komercyjnymi. RedShift Amazonu to hostowana wersja produktu firmy ParAccel. Niedawno pojawiło się wiele otwartych projektów typu „SQL w Hadoopie”. Są to nowe rozwiązania, ale mają konkurować z komercyjnymi hurtowniami danych. Do tej grupy należą: Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo i Apache Drill [52, 53]. Niektóre z nich są oparte na rozwiązaniach z systemu Dremel Google’a [54].

## Gwiazdy i płatki śniegu — schematy używane w analityce

W rozdziale 2. napisano, że w świecie przetwarzania transakcji używa się wielu różnych modeli danych dobieranych do potrzeb aplikacji. Natomiast w analityce różnorodność modeli danych jest znacznie mniejsza. Wiele hurtowni danych działa schematycznie z użyciem *schematu gwiazdy* (nazywanego też *modelem wymiarowym* [55]).

Przykładowy schemat z rysunku 3.9 pochodzi z hurtowni danych, która może być używana przez sklep spożywczy. Centrum tego schematu stanowi *tabela faktów* (w tym przykładzie jej nazwa to *fact\_sales*). Każdy wiersz tabeli faktów reprezentuje zdarzenie, które wystąpiło w określonym czasie (tu każdy wiersz reprezentuje zakup produktu przez klienta). Jeśli analizujesz ruch w witrynie, a nie transakcje w sklepie, każdy wiersz może reprezentować wyświetlenie strony lub kliknięcie elementu przez użytkownika.



Rysunek 3.9. Przykładowy schemat gwiazdy używany w hurtowni danych

Fakty to zwykle pojedyncze zdarzenia, ponieważ umożliwia to zachowanie maksymalnej elastyczności w trakcie późniejszych analiz. Jednak oznacza to też, że tabela faktów może stać się bardzo duża. W dużych firmach takich jak Apple, Walmart lub eBay hurtownia danych może obejmować dziesiątki petabajtów historii transakcji. Większość tego miejsca zajmuje tabela faktów [56].

Niektóre kolumny z tabeli faktów to atrybuty. Atrybutem jest np. cena sprzedaży produktu i zakup go od dostawcy (pozwala to obliczyć marżę zysku). Inne kolumny tabeli faktu to referencje w postaci kluczy obcych prowadzące do innych tabel nazywanych *tabelami wymiarów*. Każdy wiersz w tabeli faktów reprezentuje zdarzenia, natomiast wymiary zawierają odpowiedzi na dotyczące tych zdarzeń pytania *kto*, *co*, *gdzie*, *kiedy*, *jak* i *dlaczego*.

Na przykład na rysunku 3.9 jednym z wymiarów jest sprzedawany produkt. Każdy wiersz w tabeli `dim_product` reprezentuje jeden typ sprzedawanego produktu i obejmuje jednostkę magazynową, opis, nazwę marki, kategorię, zawartość tłuszczu, rozmiar opakowania itd. Każdy wiersz w tabeli `fact_sales` używa klucza obcego do określenia, który produkt został sprzedany w konkretnej transakcji. Dla uproszczenia przyjęto, że jeśli klient kupił kilka produktów, są one reprezentowane w tabeli faktów w odrębnych wierszach.

Nawet data i czas często są reprezentowane za pomocą tabel wymiarów, ponieważ pozwala to zakodować dodatkowe informacje na temat dat (np. o dniach wolnych od pracy). Dzięki temu w zapytaniach można wyodrębnić sprzedaż w dni wolne od pracy i sprzedaż w dni robocze.

Nazwa „schemat gwiazdy” pochodzi od tego, że w graficznym przedstawieniu relacji między tabelami tabela faktów znajduje się pośrodku i jest otoczona tabelami wymiarów. Połączenia między tabelami przypominają promienie gwiazdy.

Odmianę tego rozwiązania stanowi *schemat płatka śniegu*, w którym wymiary są dodatkowo dzielone na podwymiary. Mogą np. występować odrębne tabele z markami i kategoriami produktów, a w każdym wierszu z tabeli `dim_product` można podać markę i kategorię za pomocą kluczy obcych, zamiast zapisywać te dane jako łańcuchy znaków w tej tabeli. Schemat płatka śniegu jest znormalizowany w większym stopniu niż schemat gwiazdy, jednak ten ostatni jest często preferowany, ponieważ analitykom łatwiej z niego korzystać [55].

W typowej hurtowni danych tabele są często bardzo szerokie. Tabele faktów mają nierzadko ponad 100 kolumn, a czasem obejmują ich nawet kilkaset [51]. Tabele wymiarów także mogą być bardzo szerokie, ponieważ obejmują wszystkie metadane, które mogą być przydatne w analizach. Na przykład tabela `dim_store` może zawierać szczegółowe informacje o tym, które usługi są oferowane w poszczególnych sklepach, czy znajduje się w nim piekarnia, jaka jest powierzchnia sklepu, datę jego otwarcia, datę ostatniej rewitalizacji, odległość od najbliższej autostrady itd.

## Bazy kolumnowe

Gdy tabele faktów zawierają tryliony wierszy i petabajty danych, wydajne ich składowanie i kierowanie do nich zapytań staje się wyzwaniem. Tabele wymiarów są zwykle znacznie mniejsze (liczą miliony wierszy), dlatego w tym punkcie koncentrujemy się głównie na składowaniu faktów.

Choć tabele faktów często mają ponad 100 kolumn, typowe zapytanie do hurtowni danych używa tylko czterech lub pięciu z nich (w analityce rzadko potrzebne są zapytania `"SELECT *"`) [51]. Przyjrzyj się zapytaniu z listingu 3.1. Dotyczy ono dużej liczby wierszy (wszystkich zakupów owoców lub słodczy w 2013 r.), ale potrzebuje dostępu do tylko trzech kolumn tabeli `fact_sales`: `date_key`, `product_sk` i `quantity`. Wszystkie pozostałe kolumny są ignorowane.

*Listing 3.1. Analizowanie, czy klienci w poszczególne dni tygodnia mają większą skłonność do kupowania świeżych owoców czy słodczy*

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
JOIN dim_date ON fact_sales.date_key = dim_date.date_key
JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
```

```

WHERE
    dim_date.year = 2013 AND
    dim_product.category IN ('Świeże owoce', 'Słodkocze')
GROUP BY
    dim_date.weekday, dim_product.category;

```

Jak w wydajny sposób wykonać to zapytanie?

W większości baz OLTP dane mają układ *oparty na wierszach*. Wszystkie wartości z jednego wiersza tabeli są przechowywane obok siebie. Bazy oparte na dokumentach działają podobnie. Cały dokument jest zwykle zapisany jako sekwencja przyległych bajtów. Ilustruje to rysunek 3.1 z przykładowym plikiem CSV.

Na potrzeby przetwarzania zapytania takiego jak z listingu 3.1 można utworzyć indeksy dla kolumn `fact_sales.date_key` i/lub `fact_sales.product_sk`, informujące system składowania danych, gdzie znaleźć wszystkie transakcje dla danej daty lub określonego produktu. Jednak później system składowania danych oparty na wierszach musi wczytać wszystkie wiersze (każdy z ponad 100 atrybutami) z dysku do pamięci, przetworzyć je i odfiltrować te, które nie spełniają warunków. Może to zająć dużo czasu.

Idea działania *baz kolumnowych* jest prosta: zamiast przechowywać razem wszystkie wartości z jednego wiersza, należy zapisywać razem wszystkie wartości z każdej *kolumny*. Gdy każda kolumna jest przechowywana w odrębnym pliku, w zapytaniu trzeba tylko wczytywać i przetwarzać kolumny używane w zapytaniu, co pozwala zaoszczędzić dużo pracy. Tę zasadę przedstawia rysunek 3.10.

Tabela `fact_sales`

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Dane w układzie kolumnowym:

Zawartość pliku `date_key`: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103  
 Zawartość pliku `product_sk`: 69, 69, 69, 74, 31, 31, 31, 31  
 Zawartość pliku `store_sk`: 4, 5, 5, 3, 2, 3, 3, 8  
 Zawartość pliku `promotion_sk`: NULL, 19, NULL, 23, NULL, NULL, 21, NULL  
 Zawartość pliku `customer_sk`: NULL, NULL, 191, 202, NULL, NULL, 123, 233  
 Zawartość pliku `quantity`: 1, 3, 1, 5, 1, 3, 1, 1  
 Zawartość pliku `net_price`: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99  
 Zawartość pliku `discount_price`: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Rysunek 3.10. Przechowywanie danych relacyjnych według kolumn, a nie według wierszy



Bazy kolumnowe są najłatwiejsze do zrozumienia w modelu relacyjnym, ale równie dobrze można je stosować do danych nierelacyjnych. Na przykład Parquet [57] to format kolumnowy obsługujący model oparty na dokumentach i wzorowany na systemie Dremel Google’a [54].

Układ kolumnowy wymaga, by w każdym pliku z kolumną wiersze występowały w tej samej kolejności. Dlatego gdy chcesz ponownie uzyskać cały wiersz, możesz pobrać 23. element z każdego pliku z poszczególnymi kolumnami i scalać je w 23. wiersz tabeli.

## Kompresja kolumn

Oprócz wczytywania z dysku tylko tych kolumn, które są potrzebne w zapytaniu, można dodatkowo zmniejszyć wymogi z zakresu szybkości dysku, kompresując dane. Na szczęście bazy kolumnowe często dobrze się nadają do kompresowania.

Przyjrzyj się sekwencji wartości z każdej kolumny z rysunku 3.10. Występuje tam wiele powtórzeń, co dobrze wróży możliwościom kompresji. W zależności od danych z kolumny można stosować różne techniki kompresji. Jedną z nich, wyjątkowo skuteczną w hurtowniach danych, jest przedstawione na rysunku 3.11 *kodowanie z użyciem bitmap* (ang. *bitmap encoding*).

Wartości w kolumnie:

product\_sk: 

69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Bitmapy odpowiadające wszystkim możliwym wartościom:

product\_sk = 29: 

0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

product\_sk = 30: 

0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

product\_sk = 31: 

0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

product\_sk = 68: 

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

product\_sk = 69: 

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

product\_sk = 74: 

0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Kodowanie długości serii:

product\_sk = 29: 9, 1 (9 zer, 1 jedynka, reszta zera)

product\_sk = 30: 10, 2 (10 zer, 2 jedynki, reszta zera)

product\_sk = 31: 5, 4, 3, 3 (5 zer, 4 jedynki, 3 zera, 3 jedynki, reszta zera)

product\_sk = 68: 15, 1 (15 zer, 1 jedynka, reszta zera)

product\_sk = 69: 0, 4, 12, 2 (0 zer, 4 jedynki, 12 zer, 2 jedynki)

product\_sk = 74: 4, 1 (4 zera, 1 jedynka, reszta zera)

Rysunek 3.11. Skompresowana jedna kolumna z indeksem bitmapowym

Liczba różnych wartości w kolumnie jest często niewielka w porównaniu z liczbą wierszy (np. sklep może przeprowadzić miliardy transakcji dotyczących tylko 100 tys. różnych produktów).

Kolumnę z  $n$  różnych wartości można przekształcić w  $n$  odrębnych bitmap. Każdej różnej wartości odpowiada jedna bitmapa z jednym bitem na wiersz. Bit ma wartość 1, jeśli w wierszu znajduje się dana wartość, i 0, jeśli dana wartość nie występuje w wierszu.

Gdy  $n$  jest bardzo małe (np. w kolumnie *kraj* może się znajdować ok. 200 różnych wartości), bitmapy można zapisywać, używając jednego bita na wiersz. Jednak dla większych  $n$  w większości bitmap występować będzie bardzo dużo zer (takie bitmapy nazywa się *rzadkimi*). Wtedy można zastosować kodowanie długości serii przedstawione w dolnej części rysunku 3.11. Dzięki temu kolumnę można zapisać w niezwykle zwężony sposób.

Tego rodzaju indeksy bitmapowe świetnie nadają się do obsługi zapytań często stosowanych w hurtowniach danych. Oto przykład:

```
WHERE product_sk IN (30, 68, 69)
```

Wczytuje trzy bitmapy dla warunków `product_sk = 30`, `product_sk = 68` i `product_sk = 69` oraz wykonuje bitową operację *LUB* na trzech bitmapach, co można zrobić w bardzo wydajny sposób.

```
WHERE product_sk 31 AND store_sk = 3
```

Wczytuje bitmapy dla warunków `product_sk = 31` i `store_sk = 3` oraz wykonuje bitową operację *I*. To rozwiązanie działa, ponieważ kolumny przechowują wiersze w tej samej kolejności, dlatego  $k$ -ty bit w bitmapie dla jednej kolumny odpowiada temu samemu wierszowi co  $k$ -ty bit z bitmapy dla innej kolumny.

Istnieją też inne schematy kompresji dla różnych rodzajów danych, jednak nie będą one szczegółowo omawiane w tym miejscu. Ich przegląd znajdziesz w [58].



#### Bazy kolumnowe i rodziny kolumn

W systemach Cassandra i HBase występuje zapożyczony z systemu Bigtable mechanizm *rodzin kolumn* [9]. Jednak nazwanie tych baz „kolumnowymi” jest wysoce mylące. W ramach każdej rodziny wszystkie kolumny z wiersza są przechowywane wspólnie z kluczem wiersza, a kompresja kolumn nie jest stosowana. Tak więc model używany w systemie Bigtable jest głównie wierszowy.

### Szybkość przenoszenia danych do pamięci i przetwarzanie wektorowe

W hurtowni danych w zapytaniach, które mają skanować miliony wierszy, istotnym wąskim gardłem jest szybkość przenoszenia danych z dysku do pamięci. To jednak nie jedyne wąskie gardło. Programiści analitycznych baz danych starają się też wydajnie wykorzystać szybkość przenoszenia danych z pamięci głównej do pamięci podręcznej procesora, unikając przy tym nietrafionych przewidywań rozgałęzień i operacji pustych w potoku przetwarzania instrukcji w procesorze oraz wykorzystując architekturę **SIMD** (ang. *single-instruction-multi-data*) nowoczesnych procesorów [59, 60].

Oprócz redukcji ilości danych, jakie trzeba wczytywać z dysku, bazy kolumnowe dobrze nadają się też do wydajnego wykorzystywania cykli procesora. Na przykład system obsługi zapytań może pobrać porcję skompresowanych danych z kolumny mieszczącą się w pamięci podręcznej L1 procesora i iterować po nich w krótkiej pętli (bez wywołań funkcji). Procesor może wykonywać taką pętlę znacznie szybciej niż kod wymagający wielu wywołań funkcji i warunków dla każdego

przetwarzanego rekordu. Kompresja kolumn umożliwia zmieszczenie w tej samej pamięci podręcznej L1 większej liczby wierszy z kolumny. Operatory (takie jak wspomniane wcześniej bitowe operatory *I* oraz *OR*) można zaprojektować w taki sposób, by działały bezpośrednio na porcjach danych ze skompresowanej kolumny. Ta technika to *przetwarzanie wektorowe* [58, 49].

## Porządek sortowania w bazach kolumnowych

W bazach kolumnowych kolejność wierszy nie zawsze ma znaczenie. Najłatwiej zapisywać je w kolejności wstawiania, ponieważ wtedy wstawienie nowego wiersza oznacza dodanie danych do każdego pliku z kolumną. Można jednak zastosować określony porządek (tak jak wcześniej w plikach SSTable) i wykorzystać go do indeksowania.

Zauważ, że sortowanie każdej kolumny niezależnie od innych nie ma sensu, ponieważ wtedy nie byłoby wiadomo, które elementy z kolumn należą do tego samego wiersza. Wiersz można odtworzyć tylko dzięki temu, że wiadomo, iż *k*-ty element z jednej kolumny należy do tego samego wiersza co *k*-ty element innej kolumny.

Zamiast tego dane należy sortować według wierszy (mimo że są przechowywane kolumnowo). Administrator bazy na podstawie wiedzy o często wykonywanych zapytaniach może wybrać kolumny, według których należy posortować tabelę. Na przykład jeśli zapytania często dotyczą przedziałów czasu (takich jak ostatni miesiąc), sensowne może być użycie kolumny `date_key` jako pierwszego klucza sortowania. Następnie optymalizator zapytań może przeskanować tylko wiersze z ostatniego miesiąca, co wymaga znacznie mniej czasu niż skanowanie wszystkich wierszy.

Druga kolumna może określać kolejność wierszy mających tę samą wartość w pierwszej kolumnie. Na przykład jeśli `date_key` to pierwszy klucz sortowania danych z rysunku 3.10, właściwe może być użycie kolumny `product_sk` jako drugiego klucza sortowania, tak by wszystkie transakcje tego samego produktu z tego samego dnia były grupowane w pamięci. To pomaga w wykonywaniu zapytań, które muszą grupować lub filtrować transakcje według produktów dla określonego przedziału czasu.

Inną zaletą sortowania danych jest ułatwienie kompresji kolumn. Jeśli główna kolumna używana do sortowania nie obejmuje wielu różnych wartości, to po sortowaniu znajdują się w niej długie sekwencje z wielokrotnie powtarzającymi się tymi samymi wartościami. Proste kodowanie długości serii, takie jakie zastosowano do bitmap na rysunku 3.11, pozwala skompresować taką kolumnę do kilku kilobajtów — nawet w sytuacji, gdy tabela zawiera miliardy wierszy.

Wpływ kompresji jest największy dla pierwszego klucza sortowania. Klucze drugi i trzeci są bardziej rozproszone, dlatego nie będą obejmowały równie długich serii powtarzających się wartości. W dalszych kolumnach używanych do sortowania dane występują niemal w porządku losowym, dlatego nie pozwalają na równie skuteczną kompresję. Jednak posortowanie danych według kilku pierwszych kolumn i tak zapewnia znaczne korzyści.

### Kilka różnych porządków sortowania

Pomysłowe rozwinięcie tego pomysłu zostało wprowadzone w języku C-Store i zastosowane w komercyjnej hurtowni danych Vertica [61, 62]. W różnych zapytaniach korzystne mogą być inne porządki sortowania, dlatego więc nie przechowywać tych samych danych posortowanych na

*kilka różnych sposobów?* Dane i tak trzeba replikować na wielu maszynach, tak aby nie utracić ich w wyniku awarii jednej maszyny. Można więc równie dobrze przechowywać nadmiarowe dane posortowane w różnej kolejności, tak aby w trakcie przetwarzania zapytania można było wykorzystać najlepiej dostosowaną do niego wersję.

Stosowanie wielu porządków sortowania w bazie kolumnowej przypomina nieco używanie zestawu indeksów pomocniczych w bazie wierszowej. Ważną różnicę stanowi to, że w bazie wierszowej wszystkie wiersze są przechowywane w jednym miejscu (w pliku nieuporządkowanym lub indeksie klastrowym), a indeksy pomocnicze obejmują tylko wskaźniki do odpowiednich wierszy. W bazie kolumnowej zwykle nie występują wskaźniki prowadzące do zewnętrznych danych, a jedynie kolumny zawierające wartości.

## Zapis w bazach kolumnowych

Przedstawione optymalizacje mają sens w hurtowniach danych, ponieważ większość obciążenia w nich generują uruchamiane przez analityków długie zapytania z samym odczytem. Bazy kolumnowe, kompresja i sortowanie pomagają przyspieszyć wykonywanie zapytań z odczytem. Jednak ich wadą jest utrudnienie zapisu.

Podejście z aktualizacją w miejscu, takie jak w b-drzewach, w przypadku kolumn skompresowanych nie jest możliwe. Jeśli chcesz wstawić wiersz w środku posortowanej tabeli, zapewne będziesz musiał zapisać ponownie wszystkie pliki kolumn. Wszystkie wiersze są identyfikowane na podstawie ich pozycji w kolumnie, dlatego w trakcie wstawiania trzeba w spójny sposób zaktualizować wszystkie kolumny.

Na szczęście we wcześniejszej części rozdziału poznałeś już dobre rozwiązanie — drzewa LSM. Zapis zawsze najpierw przeprowadza się w pamięci, gdzie dane są dodawane do posortowanej struktury i przygotowywane do zapisu na dysku. Nie ma znaczenia, czy dane w pamięci są uporządkowane wierszowo czy kolumnowo. Po zakumulowaniu wystarczającej ilości danych są one scalane w plikach kolumn na dysku i zapisywane masowo w nowych plikach. Na tej zasadzie działa Vertica [62].

Zapytania muszą wtedy sprawdzać zarówno dane kolumnowe z dysku, jak i niedawne zapisy w pamięci oraz łączyć oba te źródła. Jednak optymalizator zapytań ukrywa to przed użytkownikami. Z perspektywy analityka modyfikacje danych spowodowane ich wstawianiem, aktualizowaniem lub usuwaniem są natychmiast odzwierciedlane w późniejszych zapytaniach.

## Agregowanie — kostki danych i widoki zmaterializowane

Nie każda hurtownia danych jest bazą kolumnową. Używane są też tradycyjne bazy wierszowe i inne architektury. Jednak dla doraźnych zapytań analitycznych model kolumnowy może działać znacznie szybciej, dlatego szybko zyskuje popularność [51, 63].

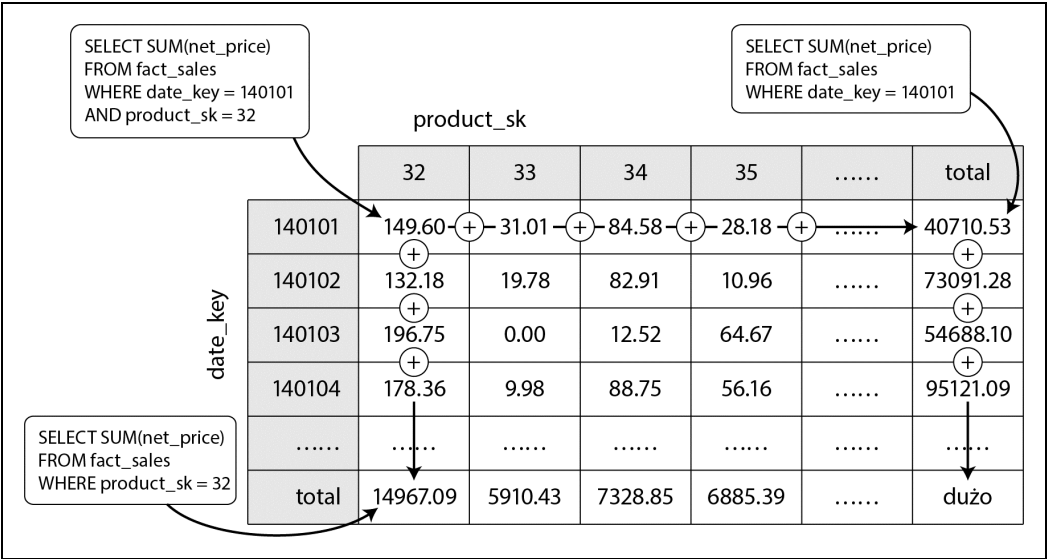
Innym aspektem hurtowni danych, o którym warto pokrótce wspomnieć, są *agregacje zmaterializowane*. Wcześniej nadmieniono, że zapytania w hurtowniach danych często obejmują funkcje agregujące (takie jak COUNT, SUM, AVG, MIN lub MAX w SQL-u). Jeśli te same agregacje są używane w wielu różnych zapytaniach, przetwarzanie za każdym razem danych surowych stanowi marnotrawstwo. Dlaczego nie zapisać w pamięci podręcznej liczb lub sum, z których zapytania korzystają najczęściej?



Jednym ze sposobów na utworzenie tego rodzaju pamięci podręcznej są *widoki zmaterializowane*. W modelu relacyjnym często definiuje się je jak widoki standardowe (wirtualne). Są to przypominające tabele obiekty, których zawartość to wynik zapytania. Różnica polega na tym, że widok zmaterializowany to zapisana na dysku kopia takich wyników, natomiast widok wirtualny to tylko skrót używany zamiast pisania zapytań. W trakcie odczytu danych z widoku wirtualnego system SQL-owy w locie rozwija widok w powiązane z nim zapytanie, a następnie wykonuje to zapytanie.

Po zmianach w danych widok zmaterializowany trzeba zaktualizować, ponieważ jest on zdenormalizowaną kopią danych. Baza może robić to automatycznie, jednak takie aktualizacje sprawiają, że zapis staje się bardziej kosztowny. Dlatego w bazach OLTP widoki zmaterializowane stosuje się rzadko. W hurtowniach danych z wieloma operacjami odczytu takie widoki są bardziej sensowne (to, czy rzeczywiście poprawiają one wydajność odczytu, zależy od konkretnej sytuacji).

Często spotykanym specjalnym rodzajem widoku zmaterializowanego jest *kostka danych* lub *kostka OLAP* [64] — to siatka agregacji pogrupowana według różnych wymiarów. Przykład pokazano na rysunku 3.12.



Rysunek 3.12. Dwa wymiary kostki danych z agregowaniem danych przez sumowanie

Wyobraź sobie, że z każdym faktem powiązane są klucze obce prowadzące do tylko dwóch tabel wymiarów. Na rysunku 3.12 są to tabele *date* i *product*. Możesz narysować dwuwymiarową tabelę z datami na jednej osi i produktami na drugiej. Każda komórka zawiera zagregowaną wartość (np. SUM) atrybutu (np. net\_price) dla wszystkich faktów o danej kombinacji data-produkt. Następnie można zastosować tę samą agregację do każdego wiersza lub kolumny i uzyskać podsumowanie zredukowane o jeden wymiar (sprzedaż dla produktów niezależnie od dat lub sprzedaż dla danej daty niezależnie od produktów).

Fakty często mają więcej niż dwa wymiary. Na rysunku 3.9 wymiarów jest pięć: data, produkt, sklep, promocja i klient. Znacznie trudniej wyobrazić sobie taką pięciowymiarową hiperkostkę, jednak zasada pozostaje taka sama: każda komórka obejmuje sprzedaż dla określonej kombinacji

daty, produktu, sklepu, promocji i klienta. Wartości można potem wielokrotnie podsumowywać według każdego wymiaru.

Zaletę zmaterializowanych kostek danych stanowi to, że niektóre zapytania można wykonywać bardzo szybko, ponieważ zostały wstępnie obliczone. Jeśli np. chcesz poznać łączną sprzedaż na sklep we wczorajszym dniu, wystarczy sprawdzić sumy w odpowiednim wymiarze. Nie trzeba skanować milionów wierszy.

Wadą jest to, że kostki danych nie zapewniają takiej elastyczności jak zapytania z użyciem danych surowych. Nie da się np. obliczyć, za jaki procent sprzedaży odpowiadają produkty droższe niż 100 zł, ponieważ cena nie jest jednym z wymiarów. Dlatego większość hurtowni danych próbuje zachowywać jak najwięcej danych surowych i stosuje agregacje takie jak kostka danych tylko do poprawy wydajności niektórych zapytań.

## Podsumowanie

Ten rozdział to próba dotarcia do istoty tego, jak bazy obsługują składowanie i pobieranie danych. Co się dzieje, gdy zapisujesz dane w bazie, i co bazy robią, gdy później żądasz tych danych?

Zobaczyłeś, że na ogólnym poziomie systemy składowania danych dzielą się na dwie szerokie kategorie: narzędzia zoptymalizowane pod kątem przetwarzania transakcji (OLTP) i pod kątem analityki (OLAP). Między wzorcami dostępu w tych dwóch obszarach występują istotne różnice:

- Systemy OLTP zwykle komunikują się z użytkownikami, co oznacza, że liczba kierowanych do nich żądań może być bardzo duża. Aby obsłużyć takie obciążenie, aplikacje zwykle w każdym zapytaniu przetwarzają tylko niewielką liczbę rekordów. Aplikacja żąda rekordów za pomocą klucza jakiegoś rodzaju, a system składowania danych używa indeksu do znajdowania danych o żądanym kluczu. Wąskim gardłem jest tu często czas wyszukiwania danych na dysku.
- Hurtownie danych i podobne systemy analityczne są mniej znane, ponieważ są używane przede wszystkim przez analityków biznesowych, a nie przez użytkowników końcowych. Obsługują one znacznie mniejszą liczbę zapytań niż systemy OLTP, jednak każde zapytanie jest zwykle bardzo wymagające i żąda przeskanowania w krótkim czasie milionów rekordów. Wąskim gardłem jest tu często szybkość dysku (a nie czas wyszukiwania), a coraz popularniejszym rozwiązaniem do obsługi takiego obciążenia roboczego są bazy kolumnowe.

W obszarze systemów OLTP opisane zostały systemy składowania danych powiązane z dwoma głównymi sposobami myślenia:

- Podejście ze strukturą dziennika, które umożliwia tylko dodawanie danych do plików i usuwanie przestarzałych plików. Zapisany plik nigdy nie jest aktualizowany. Do tej grupy należą narzędzia Bitcask, LevelDB, Cassandra, HBase, Lucene, pliki SSTable, drzewa LSM itd.
- Podejście z aktualizacją w miejscu, gdzie dysk jest traktowany jak zbiór stron o stałej wielkości, które można nadpisywać. B-drzewa są najważniejszym przykładem tego podejścia i stosuje się je we wszystkich ważnych bazach relacyjnych, a także w wielu bazach nierelacyjnych.

Systemy składowania danych ze strukturą dziennika są stosunkowo nowym rozwiązaniem. Głównym związany z nimi pomysłem jest to, aby systematycznie przekształcać zapis z dostępem losowym w zapis sekwencyjny na dysku. Pozwala to na wyższą przepustowość zapisu dzięki charakterystyce wydajności dysków twardych i dysków SSD.

Kończąc omawianie systemów OLTP, pokrótce przedstawiono bardziej złożone struktury służące do indeksowania i bazy zoptymalizowane pod kątem przechowywania wszystkich danych w pamięci.

Następnie odeszliśmy od wewnętrznych mechanizmów systemów składowania danych, aby przyjrzeć się na ogólnym poziomie architekturze typowej hurtowni danych. W ramach tego przeglądu pokazano, dlaczego obciążenie robocze związane z analityką tak bardzo się różni od modelu OLTP. Gdy zapytania wymagają sekwencyjnego skanowania dużej liczby wierszy, indeksy znacznie tracą na znaczeniu. Zamiast tego ważne staje się kodowanie danych w bardzo zwężły sposób, aby zminimalizować ilość danych, jakie zapytanie musi wczytywać z dysku. Opisano, w jaki sposób bazy kolumnowe pomagają osiągnąć ten cel.

Programista aplikacji wyposażony w przedstawioną tu wiedzę na temat wewnętrznych mechanizmów systemów składowania danych może znacznie łatwiej określić, które narzędzie jest najlepiej dostosowane do konkretnej aplikacji. Jeśli chcesz dostroić parametry bazy, ta wiedza pozwoli Ci wyobrazić sobie, jaki wpływ przyniesie zwiększenie lub zmniejszenie wartości tych parametrów.

Choć ten rozdział nie robi z Ciebie eksperta w dostrajaniu żadnego konkretnego mechanizmu składowania danych, można mieć nadzieję, że poznałeś wystarczająco dużo pojęć i rozwiązań, aby móc zrozumieć dokumentację wybranej bazy.

## **Literatura cytowana**

- [1] Alfred V. Aho, John E. Hopcroft i Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, ISBN: 978-0-201-00023-8.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest i Clifford Stein, *Introduction to Algorithms*, wydanie trzecie, MIT Press, 2009, ISBN: 978-0-262-53305-8.
- [3] Justin Sheehy i David Smith, *Bitcask: A Log-Structured Hash Table for Fast Key/Value Data*, Basho Technologies, kwiecień 2010 (<http://basho.com/wp-content/uploads/2015/05/bitcask-intro.pdf>).
- [4] Yinan Li, Bingsheng He, Robin Jun Yang i in., *Tree Indexing on Solid State Drives*, „Proceedings of the VLDB Endowment”, rocznik 3, nr 1, s. 1195 – 1206, wrzesień 2010 (<http://www.vldb.org/pvldb/vldb2010/papers/R106.pdf>).
- [5] Goetz Graefe, *Modern B-Tree Techniques*, „Foundations and Trends in Databases”, rocznik 3, nr 4, s. 203 – 402, sierpień 2011 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf>; <http://www.nowpublishers.com/article/Details/DBS-028>).
- [6] Jeffrey Dean i Sanjay Ghemawat, *LevelDB Implementation Notes*, [leveldb.googlecode.com](http://leveldb.googlecode.com) (<https://github.com/google/leveldb/blob/master/doc/impl.md>).

- [7] Dhruba Borthakur, *The History of RocksDB*, [rocksdb.blogspot.com](http://rocksdb.blogspot.com/), 24 listopada 2013 (<http://rocksdb.blogspot.com/>).
- [8] Matteo Bertozzi, *Apache HBase I/O — HFile*, [blog.cloudera.com](http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/), 29 czerwca 2012 (<http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/>).
- [9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat i in., *Bigtable: A Distributed Storage System for Structured Data*, w: „7th USENIX Symposium on Operating System Design and Implementation” (OSDI), listopad 2006 (<https://research.google.com/archive/bigtable.html>).
- [10] Patrick O’Neil, Edward Cheng, Dieter Gawlick i Elizabeth O’Neil, *The LogStructured Merge-Tree (LSM-Tree)*, „Acta Informatica”, rocznik 33, nr 4, s. 351 – 385, czerwiec 1996 (<https://www.cs.umb.edu/~poneil/lsmtree.pdf>; <https://link.springer.com/article/10.1007%2F002360050048>).
- [11] Mendel Rosenblum i John K. Ousterhout, *The Design and Implementation of a Log-Structured File System*, „ACM Transactions on Computer Systems”, rocznik 10, nr 1, s. 26 – 52, luty 1992 (<http://research.cs.wisc.edu/areas/os/Qual/papers/lfs.pdf>; <https://dl.acm.org/citation.cfm?doid=146941.146943>).
- [12] Adrien Grand, *What Is in a Lucene Index?*, w: „Lucene/Solr Revolution”, 14 listopada 2013 (<https://www.slideshare.net/gamgoster/architecture-and-implementation-of-apache-lucene-13105167>).
- [13] Deepak Kandept, *Hacking Lucene — The Index Format*, [hackerlabs.org](http://hackerlabs.org), 1 października 2011 (<http://massiveprogramming.blogspot.com/2014/10/hacking-lucene-index-format-hacker-labs.html>).
- [14] Michael McCandless, *Visualizing Lucene’s Segment Merges*, [blog.mikemccandless.com](http://blog.mikemccandless.com), 11 lutego 2011 (<http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>).
- [15] Burton H. Bloom, *Space/Time Trade-offs in Hash Coding with Allowable Errors*, „Communications of the ACM”, rocznik 13, nr 7, s. 422 – 426, lipiec 1970 (<http://www.cs.upc.edu/~diaz/p422-bloom.pdf>; <https://dl.acm.org/citation.cfm?doid=362686.362692>).
- [16] *Operating Cassandra: Compaction*, dokumentacja bazy Apache Cassandra v4.0, 2016 (<https://cassandra.apache.org/doc/latest/operating/compaction.html>).
- [17] Rudolf Bayer i Edward M. McCreight, *Organization and Maintenance of Large Ordered Indices*, Boeing Scientific Research Laboratories, „Mathematical and Information Sciences Laboratory”, raport nr 20, lipiec 1970 ([http://infolab.usc.edu/csci585/Spring2010/den\\_ar/indexing.pdf](http://infolab.usc.edu/csci585/Spring2010/den_ar/indexing.pdf)).
- [18] Douglas Comer, *The Ubiquitous B-Tree*, „ACM Computing Surveys”, rocznik 11, nr 2, s. 121 – 137, czerwiec 1979 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.6637&rep=rep1&type=pdf>; <https://dl.acm.org/citation.cfm?doid=356770.356776>).
- [19] Emmanuel Goossaert, *Coding for SSDs*, [codecapsule.com](http://codecapsule.com), 12 lutego 2014 (<http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/>).
- [20] C. Mohan i Frank Levine, *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 1992 (<https://dl.acm.org/citation.cfm?doid=130283.130338>).

- [21] Howard Chu, *LDAP at Lightning Speed*, w: „Build Stuff ’14”, listopad 2014 (<https://buildstuff14.sched.com/event/08a1a368e272eb599a52e08b4c3c779d>).
- [22] Bradley C. Kuzmaul, *A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees*, *tokutek.com*, 22 kwietnia 2014.
- [23] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas i in., *Designing Access Methods: The RUM Conjecture*, w: „19th International Conference on Extending Database Technology” (EDBT), marzec 2016 (<http://openproceedings.org/2016/conf/edbt/paper-12.pdf>).
- [24] Peter Zaitsev, „InnoDB Double Write”, *percona.com*, 4 sierpnia 2006 (<https://www.percona.com/blog/2006/08/04/innodb-double-write/>).
- [25] Tomas Vondra, *On the Impact of Full-Page Writes*, *blog.2ndquadrant.com*, 23 listopada 2016 (<https://blog.2ndquadrant.com/on-the-impact-of-full-page-writes/>).
- [26] Mark Callaghan, *The Advantages of an LSM vs a B-Tree*, *smalldatum.blogspot.co.uk*, 19 stycznia 2016 (<http://smalldatum.blogspot.co.uk/2016/01/summary-of-advantages-of-lsm-vs-b-tree.html>).
- [27] Mark Callaghan, *Choosing Between Efficiency and Performance with RocksDB*, w: „Code Mesh”, 4 listopada 2016 (<http://www.codemesh.io/codemesh2016/mark-callaghan>).
- [28] Michi Mutsuzaki, *MySQL vs. LevelDB*, *github.com*, sierpień 2011 (<https://github.com/m1ch1/mapkeeper/wiki/MySQL-vs.-LevelDB>).
- [29] Benjamin Coverston, Jonathan Ellis i in., *CASSANDRA-1608: Redesigned Compaction*, *issues.apache.org*, lipiec 2011 (<https://issues.apache.org/jira/browse/CASSANDRA-1608#>).
- [30] Igor Canadi, Siying Dong i Mark Callaghan, „RocksDB Tuning Guide”, *github.com*, 2016 (<https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>).
- [31] *MySQL 5.7 Reference Manual*, Oracle, 2014 (<https://dev.mysql.com/doc/refman/5.7/en/>).
- [32] *Books Online for SQL Server 2012*, Microsoft, 2012 (<https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation>).
- [33] Joe Webb, *Using Covering Indexes to Improve Query Performance*, *simpletalk.com*, 29 września 2008 (<https://www.red-gate.com/simple-talk/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/>).
- [34] Frank Ramsak, Volker Markl, Robert Fenk i in., *Integrating the UB-Tree into a Database System Kernel*, w: „26th International Conference on Very Large Data Bases” (VLDB), wrzesień 2000 (<http://www.vldb.org/conf/2000/P263.pdf>).
- [35] The PostGIS Development Group, *PostGIS 2.1.2dev Manual*, *postgis.net*, 2014 (<http://postgis.net/docs/manual-2.1/>).
- [36] Robert Escriva, Bernard Wong i Emin Gün Sirer, *HyperDex: A Distributed, Searchable Key-Value Store*, w: „ACM SIGCOMM Conference”, sierpień 2012 (<http://www.cs.princeton.edu/courses/archive/fall13/cos518/papers/hyperdex.pdf>; <https://dl.acm.org/citation.cfm?doid=2377677.2377681>).

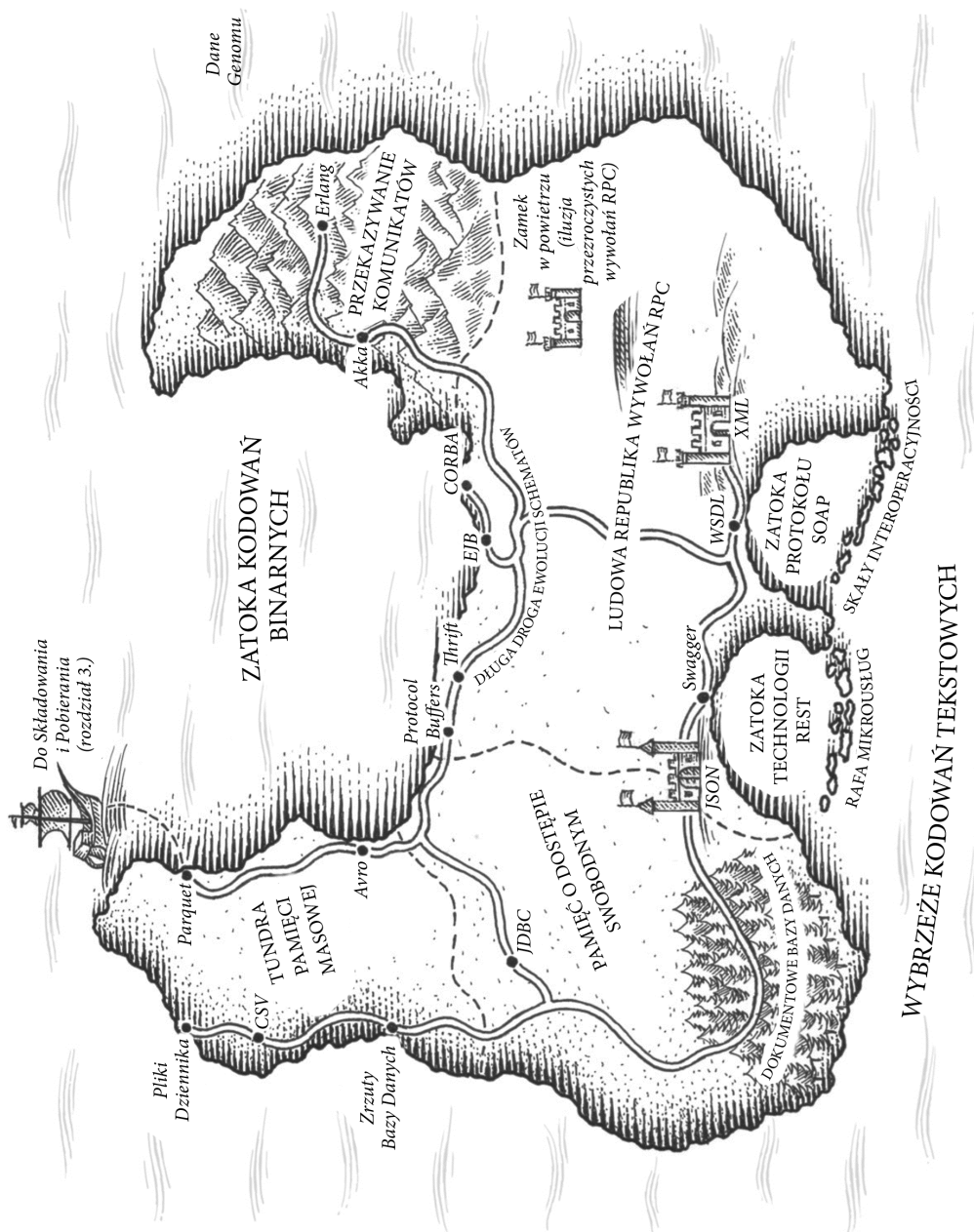
- [37] Michael McCandless, *Lucene's FuzzyQuery Is 100 Times Faster in 4.0*, *blog.mikemccandless.com*, 24 marca 2011 (<http://blog.mikemccandless.com/2011/03/lucenes-fuzzyquery-is-100-times-faster.html>).
- [38] Steffen Heinz, Justin Zobel i Hugh E. Williams, *Burst Tries: A Fast, Efficient Data Structure for String Keys*, „ACM Transactions on Information Systems”, rocznik 20, nr 2, s. 192 – 223, kwiecień 2002 (<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.3499>; <https://dl.acm.org/citation.cfm?doid=506309.506312>).
- [39] Klaus U. Schulz i Stoyan Mihov, *Fast String Correction with Levenshtein Automata*, „International Journal on Document Analysis and Recognition”, rocznik 5, nr 1, s. 67 – 85, listopad 2002 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.652>; <https://link.springer.com/article/10.1007%2Fs10032-002-0082-8>).
- [40] Christopher D. Manning, Prabhakar Raghavan i Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press, 2008, ISBN: 978-0-521-86571-5 (<https://nlp.stanford.edu/IR-book/>).
- [41] Michael Stonebraker, Samuel Madden, Daniel J. Abadi i in., *The End of an Architectural Era (It's Time for a Complete Rewrite)*, w: „33rd International Conference on Very Large Data Bases” (VLDB), wrzesień 2007 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3697&rep=rep1&type=pdf>).
- [42] „VoltDB Technical Overview White Paper”, VoltDB, 2014 (<https://www.voltdb.com/resources/>).
- [43] Stephen M. Rumble, Ankita Kejriwal i John K. Ousterhout, *Log-Structured Memory for DRAM-Based Storage*, w: „12th USENIX Conference on File and Storage Technologies” (FAST), luty 2014 (<https://www.usenix.org/node/179822>).
- [44] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden i Michael Stonebraker, *OLTP Through the Looking Glass, and What We Found There*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2008 (<http://hstore.cs.brown.edu/papers/hstore-lookingglass.pdf>; <https://dl.acm.org/citation.cfm?doid=1376616.1376713>).
- [45] Justin DeBrabant, Andrew Pavlo, Stephen Tu i in., *Anti-Caching: A New Approach to Database Management System Architecture*, „Proceedings of the VLDB Endowment”, rocznik 6, nr 14, s. 1942 – 1953, wrzesień 2013 (<http://www.vldb.org/pvldb/vol6/p1942-debrabant.pdf>).
- [46] Joy Arulraj, Andrew Pavlo i Subramanya R. Dullloor, *Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2015 (<http://www.pdl.cmu.edu/PDL-FTP/NVM/storage.pdf>).
- [47] Edgar F. Codd, S.B. Codd i C.T. Salley, *Providing OLAP to User-Analysts: An IT Mandate*, E.F. Codd Associates, 1993 ([http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem\\_dwh/lit/Cod93.pdf](http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf)).
- [48] Surajit Chaudhuri i Umeshwar Dayal, *An Overview of Data Warehousing and OLAP Technology*, „ACM SIGMOD Record”, rocznik 26, nr 1, s. 65 – 74, marzec 1997 (<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/sigrecord.pdf>; <https://dl.acm.org/citation.cfm?doid=248603.248616>).

- [49] Per-Åke Larson, Cipri Clinciu, Campbell Fraser i in., *Enhancements to SQL Server Column Stores*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2013 (<https://www.microsoft.com/en-us/research/publication/enhancements-to-sql-server-column-stores/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F193599%2Fapollo3%2520-%2520sigmod%25202013%2520-%2520final.pdf>).
- [50] Franz Färber, Norman May, Wolfgang Lehner i in., *The SAP HANA Database – An Architecture Overview*, „IEEE Data Engineering Bulletin”, rocznik 35, nr 1, s. 28 – 33, marzec 2012 (<http://sites.computer.org/debull/A12mar/hana.pdf>).
- [51] Michael Stonebraker, *The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong*, prezentacja na: EPFL, maj 2013 (<http://slideshot.epfl.ch/talks/166>).
- [52] Daniel J. Abadi, *Classifying the SQL-on-Hadoop Solutions*, *hadapt.com*, 2 października 2013 (<https://web.archive.org/web/20150622074951/http://hadapt.com/blog/2013/10/02/classifying-the-sql-on-hadoop-solutions/>).
- [53] Marcel Kornacker, Alexander Behm, Victor Bittorf i in., *Impala: A Modern, Open-Source SQL Engine for Hadoop*, w: „7th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2015 (<http://pandis.net/resources/cidr15impala.pdf>).
- [54] Sergey Melnik, Andrey Gubarev, Jing Jing Long i in., *Dremel: Interactive Analysis of Web-Scale Datasets*, w: „36th International Conference on Very Large Data Bases” (VLDB), s. 330 – 339, wrzesień 2010 (<https://research.google.com/pubs/pub36632.html>).
- [55] Ralph Kimball i Margy Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, wydanie trzecie, John Wiley & Sons, lipiec 2013, ISBN: 978-1-118-53080-1.
- [56] Derrick Harris, *Why Apple, eBay, and Walmart Have Some of the Biggest Data Warehouses You’ve Ever Seen*, *gigaom.com*, 27 marca 2013 (<https://gigaom.com/2013/03/27/why-apple-ebay-and-walmart-have-some-of-the-biggest-data-warehouses-youve-ever-seen/>).
- [57] Julien Le Dem, *Dremel Made Simple with Parquet*, *blog.twitter.com*, 11 września 2013 ([https://blog.twitter.com/engineering/en\\_us/a/2013/dremel-made-simple-with-parquet.html](https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html)).
- [58] Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos i in., *The Design and Implementation of Modern Column-Oriented Database Systems*, „Foundations and Trends in Databases”, rocznik 5, nr 3, s. 197 – 280, grudzień 2013 (<http://cs-www.cs.yale.edu/homes/dna/papers/abadi-column-stores.pdf>; <http://www.nowpublishers.com/article/Details/DBS-024>).
- [59] Peter Boncz, Marcin Zukowski i Niels Nes, *MonetDB/X100: HyperPipelining Query Execution*, w: „2nd Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2005 (<http://cidrdb.org/cidr2005/papers/P19.pdf>).
- [60] Jingren Zhou i Kenneth A. Ross, *Implementing Database Operations Using SIMD Instructions*, w: „ACM International Conference on Management of Data” (SIGMOD), s. 145 – 156, czerwiec 2002 (<http://www1.cs.columbia.edu/~kar/pubsk/simd.pdf>; <https://dl.acm.org/citation.cfm?doid=564691.564709>).

- [61] Michael Stonebraker, Daniel J. Abadi, Adam Batkin i in., *C-Store: A Column oriented DBMS*, w: „31st International Conference on Very Large Data Bases” (VLDB), s. 553 – 564, wrzesień 2005 (<http://db.csail.mit.edu/projects/cstore/vldb.pdf>).
- [62] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan i in., *The Vertica Analytic Database: C-Store 7 Years Later*, „Proceedings of the VLDB Endowment”, rocznik 5, nr 12, s. 1790 – 1801, sierpień 2012 ([http://vldb.org/pvldb/vol5/p1790\\_andrewlamb\\_vldb2012.pdf](http://vldb.org/pvldb/vol5/p1790_andrewlamb_vldb2012.pdf)).
- [63] Julien Le Dem i Nong Li, *Efficient Data Storage for Analytics with Apache Parquet 2.0*, w: „Hadoop Summit”, San Jose, czerwiec 2014 (<https://www.slideshare.net/julienledem/th-210pledem>).
- [64] Jim Gray, Surajit Chaudhuri, Adam Bosworth i in., *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*, „Data Mining and Knowledge Discovery”, rocznik 1, nr 1, s. 29 – 53, marzec 2007 (<https://arxiv.org/ftp/cs/papers/0701/0701155.pdf>; <https://link.springer.com/article/10.1023%2FA%3A1009726021843>).







WYBRZEŻE KODOWAŃ TEKSTOWYCH

# Kodowanie i zmiany

*Wszystko się zmienia i nic nie pozostaje stałe.*

— Heraklit z Efezu cytowany przez Platona w *Kratylosie* (360 r. p.n.e.)

Nieuniknione jest to, że aplikacje z czasem się zmieniają. Funkcje są dodawane lub modyfikowane z powodu udostępniania nowych produktów, lepszego zrozumienia wymagań użytkowników lub zmian w otoczeniu biznesowym. W rozdziale 1. przedstawiono kwestię *łatwości modyfikowania*. Należy dążyć do tworzenia systemów, które umożliwiają łatwe dostosowanie się do zmian (zob. punkt „Łatwość modyfikowania — umożliwianie prostego wprowadzania zmian”).

W większości przypadków modyfikacji mechanizmów oprogramowania wymaga też zmian w przechowywanych danych. Możliwe, że trzeba uwzględnić nowe pole lub nowy typ rekordu. Możliwe, że istniejące dane należy prezentować w nowy sposób.

W modelach danych omówionych w rozdziale 2. używane są różne techniki radzenia sobie z takimi zmianami. W bazach relacyjnych zwykle przyjmuje się, że wszystkie dane z bazy są zgodne z jednym schematem. Choć ten schemat może zostać zmodyfikowany (za pomocą migracji schematu, czyli z użyciem instrukcji ALTER), w każdym momencie obowiązuje tylko jeden schemat. Natomiast w bazach ze schematem określonym przy odczycie (pozbawionych schematu) nie ma obowiązującego formatu. Baza może więc obejmować mieszankę danych w starszych i nowszych formatach zapisanych w różnym czasie (zob. punkt „Elastyczność schematu w modelu opartym na dokumentach”).

Po zmianie formatu lub schematu często konieczne jest wprowadzenie odpowiednich zmian w kodzie aplikacji (np. po dodaniu do rekordu nowego pola w kodzie aplikacji trzeba zacząć wczytywać i zapisywać to pole). Jednak w dużych aplikacjach nierzadko nie da się natychmiastowo wprowadzić zmian w kodzie:

- W aplikacjach działających po stronie serwera możesz chcieć stosować *aktualizacje stopniowe* (ang. *rolling upgrade* lub *staged rollout*). Polega to na wdrażaniu nowej wersji w grupach po kilka węzłów naraz, sprawdzaniu, czy nowa wersja działa poprawnie, i stopniowym umieszczaniu jej we wszystkich węzłach. Pozwala to wdrażać nowe wersje bez przestojów usługi, co zachęca do częstszych udostępniania zmian i ułatwia modyfikowanie.
- W przypadku aplikacji działających po stronie klienta jesteś na łasce użytkowników, którzy mogą przez pewien czas nie instalować aktualizacji.

Takie sytuacje oznaczają, że stara i nowa wersja kodu oraz stare i nowe formaty danych mogą współistnieć w systemie. Aby system działał poprawnie, trzeba zachować zgodność w obu kierunkach:

#### *Zgodność wstecz*

Nowszy kod potrafi wczytywać dane zapisane przez starszy kod.

#### *Zgodność wprzód*

Starszy kod potrafi wczytywać dane zapisane przez nowszy kod.

Zgodność wstecz zwykle łatwo jest uzyskać. Autor nowszego kodu zna format danych zapisywanych przez starszy kod, dlatego może bezpośrednio dodać ich obsługę (w razie potrzeby zachowując starszy kod do wczytywania starszych danych). Uzyskanie zgodności wprzód bywa trudniejsze, ponieważ wymaga tego, by starszy kod ignorował elementy dodane w nowszych wersjach kodu.

W tym rozdziale opisano kilka formatów kodowania danych, w tym JSON, XML, Protocol Buffers, Thrift i Avro. Omawiane jest głównie to, jak te formaty uwzględniają zmiany w schemacie i jak obsługują systemy, w których muszą współistnieć stare i nowe dane oraz stary i nowy kod. Dalej opisano, jak te formaty są stosowane do składowania danych i komunikacji: w usługach internetowych, architekturze **REST** (ang. *Representational State Transfer*) i wywołaniach **RPC** (ang. *Remote Procedure Calls*), a także w systemach przekazywania komunikatów (np. z użyciem aktorów i kolejek komunikatów).

## Formaty kodowania danych

Programy zwykle korzystają z danych w (przynajmniej) dwóch różnych reprezentacjach:

1. W pamięci dane są przechowywane w obiektach, strukturach, listach, tablicach, tablicach z haszowaniem, drzewach itd. Te struktury danych są zoptymalizowane pod kątem wydajnego dostępu i wykonywania operacji z użyciem procesora (zwykle za pomocą wskaźników).
2. Gdy chcesz zapisać dane w pliku lub przesłać je siecią, musisz zakodować je jako niezależną sekwencję bajtów (np. dokument w formacie JSON). Ponieważ dla innych procesów wskaźnik nie będzie zrozumiały, reprezentacja w postaci sekwencji bajtów wygląda inaczej niż struktury danych standardowo używane w pamięci<sup>1</sup>.

Dlatego potrzebna jest technika przekształcania danych między tymi dwoma reprezentacjami. Przekształcanie z reprezentacji używanej w pamięci na sekwencję bajtów nazywa się *kodowaniem* (lub *serializowaniem*), a odwrotny proces to *dekodowanie* (lub *parsowanie* albo *deserializowanie*)<sup>2</sup>.



#### Dwuznaczna terminologia

Pojęcie *serializacja* bywa, niestety, używane także w kontekście transakcji (zob. rozdział 7.), gdzie ma zupełnie inne znaczenie. Aby uniknąć dwuznaczności, w tej książce stosuje się termin *kodowanie*, choć prawdopodobnie częściej używane jest określenie *serializacja*.

<sup>1</sup> Wyjątkiem są specjalne przypadki, np. pliki odwzorowywane w pamięci lub bezpośrednie operowanie na skompresowanych danych (co opisano w punkcie „Kompresowanie kolumn”).

<sup>2</sup> Zauważ, że *kodowanie* nie ma nic wspólnego z *szyfrowaniem*. W tej książce szyfrowanie nie jest omawiane.

Ponieważ przekształcanie to tak często spotykany problem, dostępnych jest wiele różnych bibliotek i formatów kodowania. Oto ich krótki przegląd.

## Formaty specyficzne dla języków

W wielu językach programowania występuje wbudowana obsługa kodowania obiektów z pamięci do postaci sekwencji bajtów. Na przykład Java obejmuje interfejs `java.io.Serializable` [1], Ruby bibliotekę `Marshal` [2], Python moduł `pickle` [3] itd. Istnieje też wiele niezależnych bibliotek takich jak Kryo dla Javy [4].

Biblioteki używane do kodowania są bardzo wygodne, ponieważ umożliwiają zapisywanie i odczytywanie obiektów z pamięci z użyciem minimalnej ilości dodatkowego kodu. Powodują jednak także wiele poważnych problemów:

- Kodowanie jest wtedy często powiązane z konkretnym językiem programowania, a odczyt danych w innym języku sprawia dużo trudności. Jeśli przechowujesz lub przesyłasz dane za pomocą takiego kodowania, wiążesz się z obecnie używanym językiem na potencjalnie bardzo długi czas i uniemożliwiasz integrowanie systemów z rozwiązaniami innych organizacji (gdzie mogą być używane odmienne języki).
- Aby móc odzyskać dane zapisane w tych samych typach obiektowych, proces dekodowania musi potrafić tworzyć obiekty dowolnych klas. Jest to często źródłem problemów z bezpieczeństwem [5]. Jeśli napastnik sprawi, że aplikacja zdekoduje arbitralną sekwencję bajtów, może utworzyć w ten sposób obiekty dowolnych klas, co często umożliwia mu robienie strasznych rzeczy takich jak zdalne wykonywanie dowolnego kodu [6, 7].
- W takich bibliotekach kontrola wersji danych jest często traktowana jak coś mniej istotnego. Ponieważ biblioteki te są przeznaczone do szybkiego i łatwego kodowania danych, często lekceważone są niewygodne problemy związane ze zgodnością wprzód i wstecz.
- Wydajność (czas procesora potrzebny do kodowania i dekodowania danych oraz wielkość zakodowanej struktury) także zwykła być często uznawana jako mało istotna. Na przykład wbudowana serializacja z Javy jest znana z niskiej wydajności i zajmującego dużo miejsca kodowania [8].

Z tych przyczyn zwykle zły pomysł stanowi używanie wbudowanego kodowania z języka do cokolwiek innego niż bardzo krótkotrwałe rozwiązania.

## JSON, XML i ich wersje binarne

W obszarze standardowych kodowań, które umożliwiają zapis i odczyt w wielu językach programowania, oczywistymi kandydatami są JSON i XML. Są powszechnie znane i obsługiwane oraz niemal równie powszechnie nie lubiane. XML jest często krytykowany za zbytnią rozwlekłość i niepotrzebne komplikacje [9]. Popularność formatu JSON wynika głównie z jego wbudowanej obsługi w przeglądarkach (ponieważ stanowi podzbiór JavaScriptu) i prostoty (w porównaniu z XML-em). CSV to następny popularny format niezależny od języka, oferujący jednak mniejsze możliwości.

JSON, XML i CSV to formaty tekstowe, dlatego są częściowo czytelne dla ludzi (choć ich składnia często jest tematem sporów). Obok powierzchownych problemów składniowych występują też bardziej wyrafinowane trudności:

- Kodowanie liczb jest wieloznaczne. W formatach XML i CSV nie da się odróżnić liczby od łańcucha znaków składającego się z cyfr (chyba że zastosowany zostanie zewnętrzny schemat). W JSON-ie łańcuchy znaków i liczby są odróżniane, ale liczby całkowite i zmiennoprzecinkowe są traktowane tak samo. Ponadto nie można określić precyzji liczb.

Ta ostatnia kwestia stanowi problem, gdy używane są duże wartości. Na przykład liczb całkowitych większych od 253 nie da się precyzyjnie przedstawić jako liczby zmiennoprzecinkowej o podwójnej precyzji ze standardu IEEE 754. Dlatego takie wartości stają się nieprecyzyjne, gdy są parsowane w języku używającym liczb zmiennoprzecinkowych (np. w JavaScriptcie). Liczby większe od 253 są używane np. w serwisie Twitter, gdzie do identyfikowania tweetów służą wartości 64-bitowe. Dane w formacie JSON zwracane przez interfejs API Twittera obejmują identyfikator tweetu w dwóch wersjach: jako liczbę w formacie JSON i jako łańcuch znaków z liczbą dziesiętną. Ma to na celu poradzenie sobie z tym, że aplikacje w JavaScriptcie nie parsują prawidłowo tak dużych liczb [10].

- JSON i XML zapewniają dobrą obsługę łańcuchów znaków Unicode (czyli tekstu czytelnego dla człowieka), ale nie obsługują łańcuchów binarnych (sekwencji bajtów bez kodowania znaków). Łańcuchy binarne są przydatne, dlatego użytkownicy radzą sobie z ich brakiem, kodując dane binarne jako tekst w formacie Base64. Następnie za pomocą schematu określa się, że taką wartość należy interpretować jako zakodowaną w formacie Base64. To rozwiązanie działa, jest jednak nieco przekombinowane i zwiększa rozmiar danych o 33%.
- XML [11] i JSON [12] udostępniają opcjonalną obsługę schematów. Służące do tego języki schematów dają duże możliwości, dlatego ich nauka i stosowanie sprawiają sporo trudności. Schematy XML-owe są stosowane dość często, jednak wiele narzędzi używających JSON-a ignoruje schematy. Ponieważ poprawna interpretacja danych (np. liczb i łańcuchów binarnych) zależy od informacji ze schematu, w aplikacjach, w których schematy XML-a i JSON-a nie są używane, trzeba umieścić w kodzie odpowiednią logikę kodowania i dekodowania danych.
- Format CSV nie obsługuje schematów, dlatego to w aplikacji trzeba zdefiniować znaczenie każdego wiersza i kolumny. Jeśli w wyniku zmiany w aplikacji pojawi się nowy wiersz lub kolumna, trzeba uwzględnić to ręcznie. CSV to ponadto mało jednoznaczny format. Co się stanie, jeśli wartość obejmuje przecinek lub znak nowego wiersza? Choć istnieją formalnie określone reguły używania znaków ucieczki [13], nie wszystkie parsery poprawnie je stosują.

Mimo tych wad JSON, XML i CSV są wystarczająco dobre do wielu celów. Prawdopodobnie pozostaną popularne — zwłaszcza jako formaty wymiany danych między różnymi jednostkami (np. do przesyłania danych z jednej organizacji do innej). W takim scenariuszu często nie ma znaczenia, jak elegancki lub wydajny jest dany format, o ile strony go uzgodnią. Trudność uzgodnienia *czegokolwiek* przez różne organizacje to problem poważniejszy od większości pozostałych.

## Kodowanie binarne

Gdy dane są używane tylko wewnętrznie w danej organizacji, presja na zastosowanie najprostszego wspólnego formatu kodowania jest mniejsza. Możesz np. wybrać format bardziej zwężły lub taki, który umożliwi szybsze parsowanie. W przypadku małych zbiorów danych korzyści są zwykle pomijalne, jednak gdy dojdiesz do poziomu terabajtów, wybór formatu danych może mieć istotne znaczenie.

JSON jest bardziej zwężły niż XML, ale i tak zajmuje dużo miejsca w porównaniu z formatami binarnymi. Ta obserwacja doprowadziła do powstania wielu binarnych kodowań dla formatów JSON (kilka z nich to MessagePack, BSON, BSON, BSON, BSON i Smile) i XML (np. WBXML i Fast Infoset). Te formaty znalazły zastosowania w różnych niszach, jednak żaden z nich nie jest równie powszechnie używany jak tekstowe odmiany JSON-a i XML-a.

Niektóre z wymienionych formatów rozbudowują zbiór typów danych (np. umożliwiają rozróżnienie na liczby całkowite i zmiennoprzecinkowe lub dodają obsługę łańcuchów binarnych), jednak oprócz tego nie zmieniają modelu danych używanego w JSON-ie i XML-u. Przede wszystkim z powodu tego, że nie określają schematu, muszą obejmować w zakodowanych danych wszystkie nazwy pól obiektu. Oznacza to, że w binarnym kodowaniu dokumentu w formacie JSON z listingu 4.1 trzeba gdzieś zachować łańcuchy znaków `userName`, `favoriteNumber` i `interests`.

*Listing 4.1. Przykładowy rekord, który w tym rozdziale zostanie zakodowany w kilku formatach binarnych*

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

Przyjrzyj się przykładowi zastosowania MessagePacka, binarnego kodowania dla formatu JSON. Na rysunku 4.1 pokazano sekwencję bajtów, jaką uzyskasz, jeśli zakodujesz dokument w formacie JSON z listingu 4.1 za pomocą MessagePacka [14]. Oto opis kilku pierwszych bajtów:

1. Pierwszy bajt, 0x83, oznacza, że dalej znajduje się obiekt (pierwsze cztery bity = 0x80) o trzech polach (drugie cztery bity = 0x03). Może się zastanawiasz, co się dzieje, jeśli obiekt ma więcej niż 15 pól, przez co ich liczba nie zmieści się w czterech bitach. Wtedy stosuje się inny indyktor typu, a liczba pól jest zapisywana w dwóch lub czterech bajtach.
2. Drugi bajt, 0xa8, określa, że dalej znajduje się łańcuch znaków (pierwsze cztery bity = 0xa0) o ośmiu bajtach (drugie cztery bity = 0x08).
3. Następných osiem bajtów to nazwa pola `userName` zapisana w ASCII. Ponieważ długość została podana wcześniej, nie potrzeba żadnego znacznika (ani znaku ucieczki) informującego, gdzie łańcuch się kończy.
4. Kolejnych siedem bajtów koduje sześcioliterową wartość tekstową `Martin` z przedrostkiem 0xa6 itd.

W formacie binarnym dane zajmują 66 bajtów, czyli niewiele mniej niż 81 bajtów w tekstowym kodowaniu w formacie JSON (po usunięciu białych znaków). Wszystkie kodowania binarne dla formatu JSON są pod tym względem podobne. Nie jest jasne, czy tak niewielkie zmniejszenie ilości miejsca (i czasem przyspieszenie parsowania) jest warte utraty czytelności danych dla ludzi.

W dalszych punktach zobaczysz, jak uzyskać znacznie lepszy efekt i zakodować ten sam rekord za pomocą tylko 32 bajtów.

## MessagePack

Sekwencja bajtów (66 bajtów):

83	a8	75	73	65	72	4e	61	6d	65	a6	4d	61	72	74	69	6e	ae	66	61
76	6f	72	69	74	65	4e	75	6d	62	65	72	cd	05	39	a9	69	6e	74	65
72	65	73	74	73	92	ab	64	61	79	64	72	65	61	6d	69	6e	67	a7	68
61	63	6b	69	6e	67														

Analiza:

Obiekt (3 elementy)	Tekst (o długości 8)	u s e r N a m e								Tekst (o długości 6)	M a r t i n					
83	a8	75 73 65 72 4e 61 6d 65								a6	4d 61 72 74 69 6e					
	Tekst (o długości 14)	f a v o r i t e N u m b e r														
	ae	66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72														
	uint16 1337	i n t e r e s t s														
	cd	05 39		a9	69 6e 74 65 72 65 73 74 73											
Tablica (2 elementy)	Tekst (o długości 11)	d a y d r e a m i n g														
92	ab	64 61 79 64 72 65 61 6d 69 6e 67														
	Tekst (o długości 7)	h a c k i n g														
	a7	68 61 63 6b 69 6e 67														

Rysunek 4.1. Przykładowy rekord (z listingu 4.1) zakodowany za pomocą formatu MessagePack

## Thrift i Protocol Buffers

Apache Thrift [15] i Protocol Buffers (protobuf) [16] to biblioteki do kodowania binarnego oparte na tej samej zasadzie. Protocol Buffers została pierwotnie opracowana w Google’u, a Thrift — w Facebooku. Obie zostały udostępnione jako narzędzia otwarte w latach 2007 – 2008 [17].

I Thrift, i Protocol Buffers wymagają schematu kodowanych danych. Aby zakodować dane z listingu 4.1 za pomocą Thrifta, należy opisać schemat w języku **IDL** (ang. *Interface Definition Language*) tego narzędzia:

```
struct Person {  
    1: required string      userName,  
    2: optional i64        favoriteNumber,  
    3: optional list<string> interests  
}
```

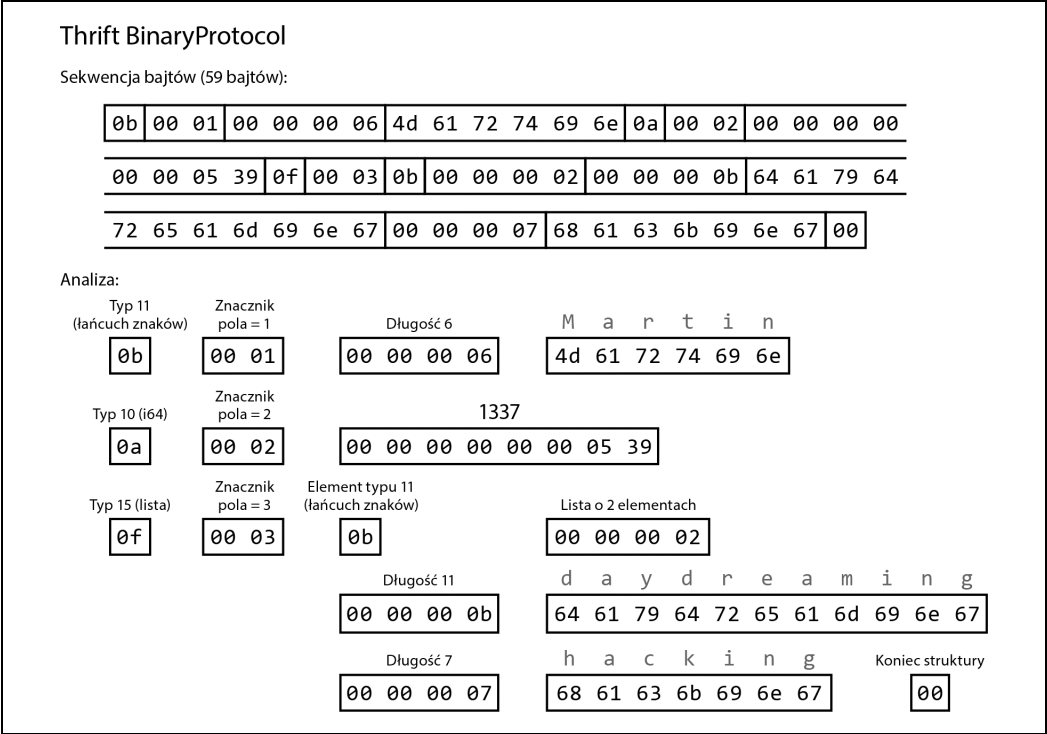
Analogiczna definicja schematu w przypadku mechanizmu Protocol Buffers wygląda bardzo podobnie:



```
message Person {
  required string user_name      = 1;
  optional int64  favorite_number = 2;
  repeated string interests      = 3;
}
```

W przypadku mechanizmów Thrift i Protocol Buffers dostępne są narzędzia do generowania kodu, które przyjmują definicję schematu podobną do pokazanych wcześniej i generują klasy z implementacją tego schematu w różnych językach programowania [18]. Aplikacja może wywoływać ten wygenerowany kod w celu kodowania lub dekodowania zgodnych ze schematem rekordów.

Jak wyglądają dane zakodowane zgodnie ze schematem? Co dziwne, Thrift używa dwóch różnych formatów kodowania binarnego<sup>3</sup> nazywanych *BinaryProtocol* i *CompactProtocol*. Przyjrzyj się najpierw formatowi *BinaryProtocol*. Zakodowane w tym formacie dane z listingu 4.1 zajmują 59 bajtów, co przedstawia rysunek 4.2 [19].



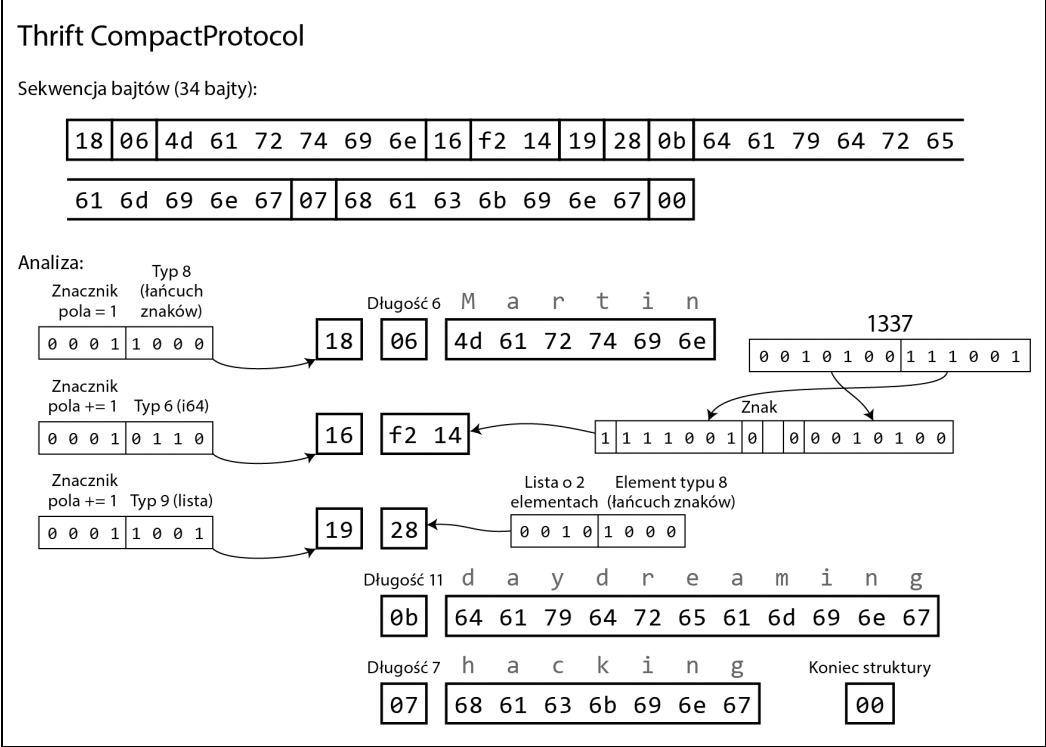
Rysunek 4.2. Przykładowy rekord zakodowany za pomocą formatu *BinaryProtocol* z Thrifta

Podobnie jak na rysunku 4.1 dla każdego pola określone są: typ (informujący, czy dane to łańcuch znaków, liczba całkowita, lista itd.) i, jeśli to potrzebne, długość (długość łańcucha znaków, liczba elementów na liście itd.). Występujące w danych łańcuchy znaków („Martin”, „daydreaming”, „hacking”) są, podobnie jak wcześniej, zakodowane w ASCII (a nie w UTF-8).

<sup>3</sup> Tak naprawdę dostępne są trzy formaty (*BinaryProtocol*, *CompactProtocol* i *DenseProtocol*), choć *DenseProtocol* jest obsługiwany tylko dla języka C++, dlatego nie jest międzyjęzykowy [18]. Ponadto istnieją dwa różne formaty kodowania oparte na JSON-ie [19]. Co za zabawa!

Istotną różnicą w porównaniu z rysunkiem 4.1 jest brak nazw pól (userName, favoriteNumber, interests). Zamiast tego zakodowane dane obejmują *znaczniki pól*, którymi są liczby (1, 2 i 3). Są to wartości z definicji schematu. Te znaczniki działają jak aliasy pól. Są zwięzłym sposobem na określenie, które pole jest opisywane, bez konieczności podawania nazwy tego pola.

Kodowanie CompactProtocol z Thrifta stanowi semantyczny odpowiednik kodowania BinaryProtocol, jednak jak widać na rysunku 4.3, mieści te same informacje w tylko 34 bajtach. Dzieje się tak dzięki umieszczeniu typu pola i znacznika w jednym bajcie oraz zastosowaniu liczb całkowitych o zmiennej długości. Zamiast przeznaczać całe osiem bajtów na liczbę 1337, jest ona kodowana za pomocą dwóch bajtów, a pierwszy bit każdego bajta informuje, czy liczba obejmuje także dalsze bajty. To oznacza, że wartości z przedziału od -64 do 63 są zakodowane w jednym bajcie, liczby od -8192 do 8191 w dwóch bajtach itd. Większe liczby zajmują więcej bajtów.



Rysunek 4.3. Przykładowy rekord zakodowany za pomocą formatu CompactProtocol z Thrifta

Protocol Buffers (udostępniający tylko jeden format kodowania binarnego) koduje te same dane w sposób pokazany na rysunku 4.4. Tu bity są rozmieszczone trochę inaczej, ale oprócz tego rozwiązanie to bardzo przypomina format CompactProtocol Thrifta. Protocol Buffers mieści ten sam rekord w 33 bajtach.

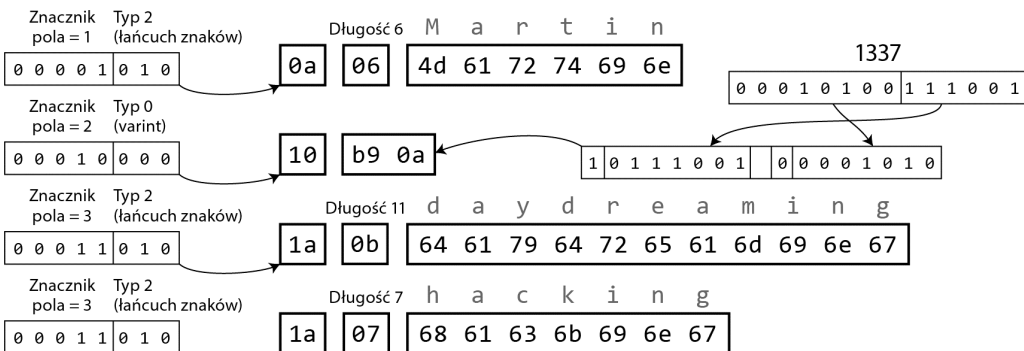
Warto zwrócić uwagę na pewien szczegół — w przedstawionych wcześniej schematach każde pole zostało oznaczone jako wymagane (required) lub opcjonalne (optional). Nie ma to jednak wpływu na to, jak pole jest kodowane (nic w danych binarnych nie informuje, czy pole jest wymagane).

## Protocol Buffers

Sekwencja bajtów (33 bajty):

0a	06	4d	61	72	74	69	6e	10	b9	0a	1a	0b	64	61	79	64	72	65	61
6d	69	6e	67	1a	07	68	61	63	6b	69	6e	67							

Analiza:



Rysunek 4.4. Przykładowy rekord zakodowany mechanizmu Protocol Buffers

Różnica polega tylko na tym, że pola wymagane umożliwiają przeprowadzenie w czasie wykonywania programu testu. Ten test kończy się niepowodzeniem, jeśli pole nie zostało ustawione. Może to być przydatne do wykrywania błędów.

### Znaczniki pól i ewolucja schematu

Wcześniej stwierdzono, że zmiany schematu w czasie są nieuniknione. Ten proces nazywa się *ewolucją schematu*. W jaki sposób Thrift i Protocol Buffers obsługują zmiany schematu, zachowując przy tym zgodność wstecz i wprzód?

Z przykładów wynika, że zakodowany rekord to tylko połączone zakodowane pola. Każde pole jest identyfikowane za pomocą znacznika (liczby 1, 2 i 3 z przykładowych schematów) i opatrzone typem danych (np. łańcuch znaków lub liczba całkowita). Jeśli nie podano wartości pola, zostaje ono pominięte w zakodowanym rekordzie. Wynika z tego, że znaczniki pól są niezbędne do określania znaczenia zakodowanych danych. Możesz zmienić nazwę pola w schemacie, ponieważ zakodowane dane nigdy nie obejmują nazw pól. Nie można jednak zmienić znacznika pola, ponieważ sprawi to, że wszystkie istniejące zakodowane dane staną się nieprawidłowe.

Możesz dodawać do schematu nowe pola, pod warunkiem że każdemu z nich przypiszesz nowy znacznik. Gdy starszy kod (w którym nowe znaczniki nie są znane) będzie próbował wczytać dane zapisane przez nowy kod, obejmujące nowe pole z niezrozumiałym znacznikiem, może zignorować dane pole. Adnotacje określające typ umożliwiają parserowi ustalenie, ile bajtów powinien pominąć. To pozwala zachować zgodność wprzód — starszy kod nadal może wczytywać rekordy zapisywane przez nowszy kod.

A co ze zgodnością wstecz? Dopóki każde pole ma unikatowy znacznik, nowy kod zawsze może wczytywać starsze dane, ponieważ znaczniki wciąż mają to samo znaczenie. Jedyne problemy polegają na tym, że gdy dodasz nowe pole, nie możesz utworzyć go jako wymaganego. Jeśli dodasz nowe pole jako wymagane, to gdy nowy kod będzie wczytywał dane zapisane przez jego starszą wersję, sprawdzanie danych zakończy się niepowodzeniem. Wynika to z tego, że starszy kod nie utworzył dodanego nowego pola. Dlatego w celu zachowania zgodności wstecz każde pole dodane po początkowym wdrożeniu schematu musi być opcjonalne lub mieć wartość domyślną.

Usuwanie pola odbywa się jak jego dodawanie, przy czym zagadnienia zgodności wstecz i wprzód są tu odwrócone. Oznacza to, że możesz usunąć pole tylko wtedy, jeśli jest opcjonalne (pól wymaganych nigdy nie można usuwać), a ponadto nigdy nie można ponownie wykorzystać stosowanego wcześniej znacznika (ponieważ gdzieś zapisane mogą być obejmujące dawny znacznik pola, które muszą być ignorowane w nowym kodzie).

## Typy danych a ewolucja schematu

A co ze zmianą typu danych pola? Jest to możliwe (szczegółowe informacje znajdziesz w dokumentacji), jednak istnieje ryzyko utraty precyzji lub przycięcia wartości. Załóżmy, że 32-bitową liczbę całkowitą zmieniasz w 64-bitową. Nowy kod może wczytywać dane zapisane przez starszy, ponieważ parser może uzupełnić brakujące bity zerami. Jednak gdy starszy kod wczytuje dane zapisane przez nowszy, nadal używa 32-bitowej zmiennej do zapisywania wartości. Jeśli zdekodowana wartość 64-bitowa nie mieści się w 32 bitach, zostanie przycięta.

Ciekawym szczegółem mechanizmu Protocol Buffers może być to, że nie obejmuje on list ani tablic. Zamiast nich dla pól używany jest znacznik `repeated` (czyli „powtarzane”; to trzecia opcja obok `required` i `optional`). Na rysunku 4.4 widać, że pole typu `repeated` działa zgodnie z jego nazwą — ten sam znacznik wielokrotnie pojawia się w rekordzie. Ma to ten wygodny skutek, że pole `optional` (o jednej wartości) można zmienić w pole `repeated` (o wielu wartościach). Nowszy kod wczytujący stare dane widzi wtedy listę z zerem elementów lub z jednym elementem (w zależności od tego, czy dane pole wcześniej istniało). Starszy kod wczytujący nowe dane widzi tylko ostatni element listy.

Thrift udostępnia specjalny typ listowy, którego parametrem jest typ danych elementów listy. To rozwiązanie nie pozwala na możliwą w mechanizmie Protocol Buffers zmianę pól z jedną wartością na pola wielowartościowe, ma jednak tę zaletę, że obsługuje listy zagnieżdżone.

## Avro

Apache Avro [20] to następny format binarny. Różni się on w ciekawy sposób od formatów Protocol Buffers i Thrift. Avro powstał w 2009 r. jako podprojekt Hadoopa, ponieważ Thrift okazał się niewystarczająco dostosowany do przypadków użycia Hadoopa [21].

W Avro do określania struktury kodowanych danych także używany jest schemat. Stosowane są tu dwa języki schematów. Jeden (Avro IDL) jest przeznaczony do edytowania kodu przez ludzi, drugi (oparty na JSON-ie) jest bardziej czytelny dla maszyn.

Przykładowy schemat zapisany w języku Avro IDL może wyglądać tak:

```

record Person {
  string          userName;
  union { null, long } favoriteNumber = null;
  array<string>   interests;
}

```

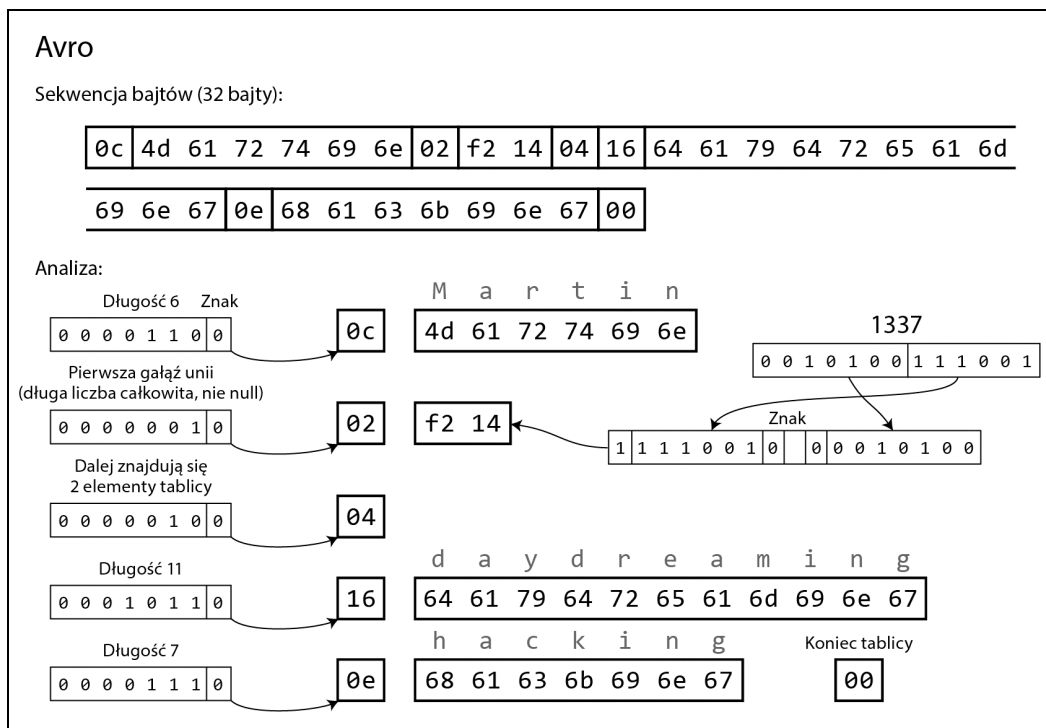
A oto analogiczna reprezentacja w JSON-ie:

```

{
  "type": "record",
  "name": "Person",
  "fields": [
    { "name": "userName",      "type": "string" },
    { "name": "favoriteNumber", "type": ["null", "long"], "default": null },
    { "name": "interests",    "type": { "type": "array", "items": "string" } }
  ]
}

```

Przed wszystkim zwróć uwagę na to, że w tym schemacie nie występują znaczniki. Jeśli za pomocą tego schematu zakodujesz przykładowy rekord (z listingu 4.1), w postaci binarnej w Avro dane zajmą tylko 32 bajty. To najbardziej zwięźle z przedstawionych kodowań. Analizę zakodowanej sekwencji bajtów pokazano na rysunku 4.5.



Rysunek 4.5. Przykładowy rekord zakodowany za pomocą Avro

Gdy przyjrzyysz się sekwencji bajtów, zauważysz, że nie ma w niej niczego, co identyfikuje pola lub ich typy danych. To kodowanie jedynie obejmuje powiązane ze sobą wartości. Łańcuch znaków to tylko określający długość przedrostek, po którym następują bajty ze znakami UTF-8. Nic w zakodowanych danych nie informuje, że używany jest łańcuch znaków. Równie dobrze mogłaby to

być liczba całkowita lub coś zupełnie innego. Liczby całkowite są kodowane za pomocą różnej liczby bajtów (tak samo jak w formacie CompactProtocol w Thrifcie).

W trakcie parsowania tych danych binarnych należy pobierać pola w kolejności ich występowania w schemacie i wykorzystać schemat do określenia typu danych każdego pola. To oznacza, że dane binarne można poprawnie zdekodować wyłącznie wtedy, gdy kod wczytujący dane używa *tego samego schematu* co kod, który je zapisał. Jakikolwiek niedopasowanie schematów między kodem wczytującym a zapisującym skutkuje nieprawidłowo zdekodowanymi danymi.

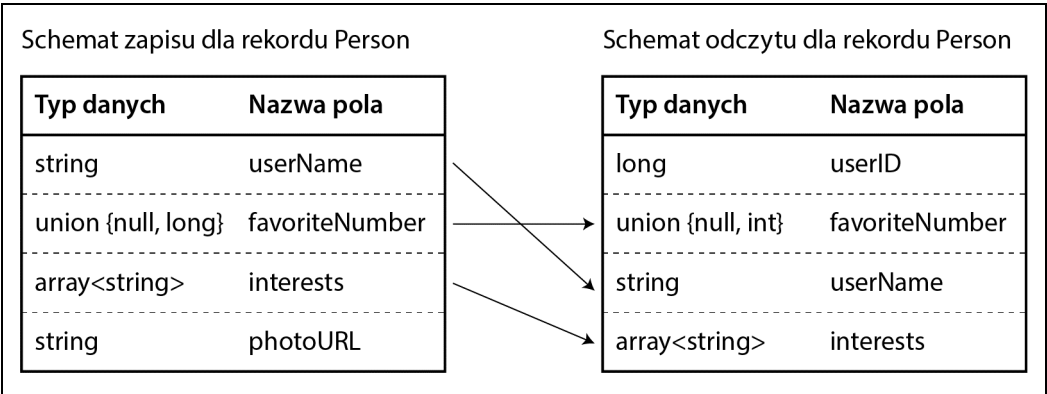
W jaki więc sposób Avro umożliwia ewolucję schematu?

**Schemat zapisu i schemat odczytu**

Gdy aplikacja używa systemu Avro i chce zakodować dane (aby zapisać je w pliku lub bazie, przesłać w sieci itd.), koduj je za pomocą znanej jej wersji schematu. Schemat może być np. skompilowany razem z aplikacją. To tzw. *schemat zapisu*.

Kiedy aplikacja chce zdekodować dane (wczytuje je z pliku lub bazy, odbiera z sieci itd.), oczekuje, że będą one miały określony schemat. To tzw. *schemat odczytu*. Jest to schemat, na którym polega kod aplikacji — kod mógł zostać wygenerowany zgodnie z tym schematem w procesie budowania aplikacji.

W systemie Avro najważniejsze jest to, że schemat zapisu i schemat odczytu *nie muszą być takie same*. Muszą tylko być zgodne ze sobą. Gdy dane są dekodowane (wczytywane), biblioteka Avro eliminuje różnice, analizując jednocześnie schemat zapisu i schemat odczytu oraz przekształcając dane ze schematu zapisu na schemat odczytu. W specyfikacji systemu Avro [20] ten proces został ściśle zdefiniowany. Ilustruje go rysunek 4.6.



Rysunek 4.6. Jednostka odczytująca w systemie Avro eliminuje różnice między schematami zapisu i odczytu

Na przykład występowanie pól w schematach zapisu i odczytu w innej kolejności nie stanowi problemu, ponieważ proces uzgadniania schematu dopasowuje pola na podstawie ich nazw. Jeśli kod wczytujący dane natrafi na pole, które występuje w schemacie zapisu, ale jest nieobecne w schemacie odczytu, ignoruje je. Jeżeli kod wczytujący dane oczekuje jakiegoś pola, ale w schemacie zapisu pole o danej nazwie nie występuje, używana jest wartość domyślna zadeklarowana w schemacie odczytu.

## Reguły ewolucji schematów

W Avro zgodność wprzód oznacza, że jednostka zapisująca może używać nowej wersji schematu, a jednostka odczytująca starej. Zgodność wstecz polega na tym, że jednostka odczytująca może używać nowej wersji schematu, a jednostka zapisująca — starej.

Aby zachować zgodność, można dodawać lub usuwać wyłącznie pola z wartością domyślną. Pole `favoriteNumber` w pokazanym schemacie Avro ma wartość domyślną `null`. Załóżmy, że dodajesz pole z wartością domyślną. Istnieje ono w nowym schemacie, ale nie ma go w starym. Gdy jednostka odczytująca używa tego schematu i wczytuje rekord zapisany za pomocą starszego schematu, brakujące pole jest uzupełniane wartością domyślną.

Gdybyś dodał pole bez wartości domyślnej, nowe jednostki odczytujące nie mogłyby wczytywać danych zapisanych przez starsze jednostki zapisujące. Oznaczałoby to naruszenie zgodności wstecz. Z kolei usunięcie pola bez wartości domyślnej powoduje, że starsze jednostki odczytujące nie mogą wczytać danych zapisanych przez nowe jednostki zapisujące. Oznacza to naruszenie zgodności wprzód.

W niektórych językach programowania `null` to akceptowalna wartość domyślna dowolnej zmiennej. W Avro jest inaczej. Jeśli chcesz umożliwić stosowanie dla pola wartości `null`, musisz zastosować typ `union`. Na przykład kod `union { null, long, string } field;` oznacza, że pole `field` może zawierać liczbę, łańcuch znaków lub wartość `null`. `Null` można stosować jako wartość domyślną tylko wtedy, gdy jest ona jedną z gałęzi typu `union`<sup>4</sup>. To nieco bardziej rozwlekłe niż domyślna możliwość przypisywania wartości `null` do wszystkich elementów, jednak pomaga zapobiegać błędom dzięki bezpośredniemu określaniu, co może być równe `null`, a co nie może [22].

Dlatego w Avro nie występują znaczniki `optional` i `required` znane z formatów Protocol Buffers i Thrift (zamiast nich używane są typ `union` i wartości domyślne).

Zmiana typu danych jest możliwa, ale pod warunkiem że Avro potrafi przekształcić dane między typami. Zmiana nazwy pola jest dopuszczalna, choć dość skomplikowana. Schemat odczytu może obejmować aliasy nazw pól, dlatego można dopasować do nich nazwy pól z dawnego schematu zapisu. To oznacza, że zmiana nazwy pola jest zgodna wstecz, ale już nie wprzód. Podobnie dodanie gałęzi do typu `union` jest zgodne wstecz, ale nie wprzód.

## Jak jednak wygląda schemat zapisu?

Pojawia się ważne pytanie, które do tego miejsca było ignorowane: skąd jednostka odczytująca zna schemat zapisu, za pomocą którego zakodowano określone dane? Nie można dołączać całego schematu do każdego rekordu, ponieważ schemat byłby dłuższy niż zakodowane dane, przez co oszczędność miejsca uzyskana dzięki kodowaniu binarnemu poszłaby na marne.

Rozwiązanie zależy od kontekstu użytkowania systemu Avro. Oto kilka przykładów:

*Duży plik z dużą liczbą rekordów*

Częstym zastosowaniem Avro (zwłaszcza w kontekście Hadoopa) jest przechowywanie dużego pliku zawierającego miliony rekordów zakodowanych za pomocą tego samego schematu

---

<sup>4</sup> Ścisłe rzecz biorąc, wartość domyślna musi być typu *pierwszej* gałęzi z unii, przy czym jest to ograniczenie specyficzne dla Avro, a nie ogólna cecha unii.

(sytuację tego rodzaju opisano w rozdziale 10.). Wtedy jednostka zapisująca może umieścić schemat zapisu raz na początku pliku. W Avro określony jest służący do tego format pliku (obiektywne pliki kontenerowe).

#### *Baza danych z pojedynczo zapisywanymi rekordami*

W bazie poszczególne rekordy mogą być zapisywane w różnych momentach za pomocą rozmaitych schematów zapisu. Nie można zakładać, że wszystkie rekordy mają ten sam schemat. Najprostsze rozwiązanie polega na dołączeniu numeru wersji na początku każdego zakodowanego rekordu i przechowywaniu listy wersji schematu w bazie. Jednostka odczytująca może wczytać rekord, ustalić numer wersji, a następnie pobrać z bazy schemat zapisu odpowiadający tej wersji. Na podstawie schematu zapisu może następnie zdekodować resztę rekordu. Działa tak np. Espresso [23].

#### *Przesyłanie rekordów przez połączenie sieciowe*

Gdy dwa procesy komunikują się przez dwukierunkowe połączenie sieciowe, mogą ustalać wersję schematu w trakcie nawiązywania połączenia, a następnie przez czas jego trwania posługiwać się tym schematem. Tak działa protokół RPC w Avro (zob. punkt „Przepływ danych w usługach: REST i RPC”).

Baza z wersjami schematu jest przydatna niezależnie od sytuacji, ponieważ pełni funkcję dokumentacji i umożliwia sprawdzenie zgodności schematów [24]. Jako numer wersji można stosować zwykle rosnące liczby całkowite lub skróty schematów.

### **Schematy generowane dynamicznie**

Jedną z zalet podejścia stosowanego w Avro (w porównaniu z mechanizmami Protocol Buffers i Thrift) jest to, że schemat nie zawiera żadnych znaczników. Dlaczego ma to znaczenie? Jakim problemem może być przechowywanie w schemacie kilku liczb?

Liczy się to, że Avro ułatwia stosowanie schematów *generowanych dynamicznie*. Załóżmy, że istnieje baza relacyjna, której zawartość zamierzasz zapisać w pliku. Chcesz zastosować format binarny, aby uniknąć opisanych wcześniej problemów z formatami tekstowymi (takimi jak JSON, CSV i SQL). Jeśli używasz Avro, możesz na podstawie schematu relacyjnego stosunkowo łatwo wygenerować schemat Avro (z opisaną wcześniej reprezentacją w formacie JSON), zakodować zawartość bazy za pomocą tego schematu i umieścić wszystkie dane w obiektywnym pliku kontenerowym Avro [25]. Należy wygenerować schemat rekordu dla każdej tabeli z bazy, a każda kolumna bazy odpowiada polu rekordu. Nazwy kolumn z bazy odpowiadają nazwom pól w Avro.

Jeśli schemat bazy się zmieni (np. w tabeli dodana zostanie jedna kolumna i usunięta inna), można wygenerować nowy schemat Avro na podstawie zaktualizowanego schematu bazy i wyeksportować dane w nowym schemacie Avro. W procesie eksportowania danych nie trzeba zwracać uwagi na zmiany w schemacie. Wystarczy przeprowadzić konwersję schematu przy każdym uruchamianiu tego procesu. Każdy, kto wczytuje nowe pliki z danymi, zobaczy, że pola w rekordzie zostały zmienione. Jednak ponieważ pola są identyfikowane na podstawie nazw, zaktualizowany schemat zapisu nadal można dopasować do starszego schematu odczytu.



Z kolei zastosowanie w tym kontekście mechanizmów Thrift lub Protocol Buffers spowoduje, że znaczniki pól zapewne trzeba będzie zmodyfikować ręcznie. Po każdej zmianie schematu bazy administrator będzie musiał ręcznie zaktualizować odwzorowania nazw kolumn bazy na znaczniki pól. Czasem ten proces można zautomatyzować, jednak generator schematu musi zachować dużą ostrożność, aby nie przypisać wcześniej używanych znaczników pól. Tego rodzaju dynamicznie generowane schematy nie były celem projektowym w trakcie tworzenia mechanizmów Thrift lub Protocol Buffers, natomiast zostały uwzględnione w Avro.

### Generowanie kodu i języki z dynamicznym określaniem typów

Thrift i Protocol Buffers wykorzystują generowanie kodu. Po zdefiniowaniu schematu można wygenerować kod implementujący ten schemat w wybranym języku programowania. Jest to przydatne w językach ze statycznym określaniem typów, np. w Javie, C++ lub C#, ponieważ w przypadku zdekodowanych danych można wykorzystać wydajne struktury działające w pamięci, a w środowiskach IDE obsługuje się sprawdzanie typów i automatyczne uzupełnianie kodu w trakcie pisania programów używających tych struktur.

W językach z dynamicznym określaniem typów, np. w JavaScriptcie, Ruby i Pythonie, generowanie kodu nie ma dużego sensu, ponieważ nie występuje w nich kontrola typów na etapie kompilacji. W tych językach generowanie kodu jest często źle widziane, gdyż unika się w nich bezpośredniego etapu kompilacji. Ponadto gdy stosowane są dynamicznie generowane schematy (np. schemat Avro generowany na podstawie tabeli bazy danych), generowanie kodu stanowi niepotrzebną przeszkodę na drodze do otrzymania danych.

Avro umożliwia opcjonalne generowanie kodu dla języków programowania ze statycznym określaniem typów, jednak ten system działa też bez generowania kodu. Jeśli istnieje obiektowy plik kontenerowy (obejmujący schemat zapisu), możesz go otworzyć za pomocą biblioteki Avro i przejrzeć dane w taki sam sposób jak w pliku w formacie JSON. Obiektowy plik kontenerowy jest *samoopisowy*, ponieważ obejmuje wszystkie potrzebne metadane.

Ta cecha przydaje się zwłaszcza w połączeniu z językami przetwarzania danych z dynamicznym określaniem typów, np. z językiem Apache Pig [26]. W języku Pig możesz otworzyć pliki Avro, zacząć je analizować i zapisać uzyskane zbiory danych do plików wyjściowych w formacie Avro, nie martwiąc się przy tym o schematy.

### Zalety schematów

Wiesz już, że Protocol Buffers, Thrift i Avro używają schematu do opisu formatu kodowania binarnego. Język schematów w tych narzędziach jest znacznie prostszy niż XML Schema lub JSON Schema, które obsługują dużo bardziej szczegółowe reguły sprawdzania poprawności (np. „tekst z tego pola musi pasować do tego wyrażenia regularnego” lub „liczba całkowita z tego pola musi być z przedziału między 0 a 100”). Ponieważ Protocol Buffers, Thrift i Avro są prostsze w implementacji i w użyciu, obsługują spory zakres języków programowania.

Pomysły, na których oparto te kodowania, nie są nowe. Mają one np. dużo wspólnego z ASN.1, językiem definiowania schematów, który po raz pierwszy ustandaryzowano w 1984 r. [27]. Służył on do definiowania różnych protokołów sieciowych, a stosowane w nim kodowanie binarne (DER)

nadal jest używane np. do kodowania certyfikatów SSL (X.509). ASN.1 obsługuje ewolucję schematów z użyciem znaczników (podobnie jak Protocol Buffers i Thrift) [29]. Język ten jest jednak bardzo złożony i słabo udokumentowany, dlatego w nowych aplikacjach zwykle nie stanowi dobrego wyboru.

W wielu systemach danych stosowane są zastrzeżone kodowania binarne. Na przykład w większości baz relacyjnych używa się protokołu sieciowego pozwalającego przysyłać zapytania do bazy i otrzymywać odpowiedzi. Takie protokoły są zwykle specyficzne dla konkretnej bazy, a producent bazy udostępnia sterownik (używający np. interfejsów ODBC lub JDBC), który dekoduje odpowiedzi z protokołu sieciowego bazy na struktury danych używane w pamięci.

Tak więc choć tekstowe formaty danych takie jak JSON, XML i CSV są powszechnie stosowane, kodowania binarne oparte na schematach też są akceptowalnym rozwiązaniem. Takie kodowania mają wiele przydatnych cech:

- Mogą być dużo bardziej zwarte niż różne odmiany „binarnego formatu JSON”, ponieważ w zakodowanych danych można pominąć nazwy pól.
- Schemat to przydatna forma dokumentacji, a ponieważ jest potrzebny do dekodowania, można mieć pewność, że jest aktualny (ręcznie aktualizowana dokumentacja może być niezgodna z rzeczywistością).
- Utrzymywanie bazy schematów sprawia, że przed wdrożeniem jakichkolwiek zmian w schemacie można sprawdzić, czy są one zgodne wprzód i wstecz.
- Dla użytkowników języków ze statycznym określaniem typów przydatna jest możliwość generowania kodu na podstawie schematu, ponieważ pozwala to na sprawdzanie typów w czasie kompilacji.

Oto podsumowanie: ewolucja schematów zapewnia tego samego rodzaju elastyczność co bazy pozbawione schematu lub ze schematem z etapu odczytu dla danych w formacie JSON (zob. punkt „Elastyczność schematów w modelu opartym na dokumentach”), a jednocześnie oferuje większe gwarancje dotyczące danych i lepsze narzędzia.

## Sposoby przepływu danych

Na początku rozdziału stwierdzono, że gdy chcesz przesłać dane między procesami, które nie współużytkują tej samej pamięci (np. w celu przesłania danych przez sieć lub zapisania ich w pliku), musisz zakodować je jako sekwencję bajtów. Dalej opisano różne służące do tego kodowania.

Omówiono też zgodność wprzód i wstecz. Jest ona istotna w kontekście łatwości modyfikowania (ułatwiania wprowadzania zmian w formie niezależnej aktualizacji różnych części systemu bez konieczności modyfikowania wszystkiego w jednym kroku). Zgodność wynika z relacji między jednym procesem (kodującym dane), a drugim (dekodującym je).

To dość abstrakcyjna idea. Istnieje wiele sposobów przepływu danych z jednego procesu do innego. Co koduje dane, a co je odkodowuje? W dalszej części rozdziału opisane zostały najczęściej spotykane sposoby przepływu danych między procesami:

- z użyciem baz danych (zob. punkt „Przepływ danych z wykorzystaniem baz”);
- za pomocą wywołań usługi (zob. punkt „Przepływ danych z użyciem usług: REST i RPC”);
- z wykorzystaniem asynchronicznego przekazywania komunikatów (zob. punkt „Przepływ danych za pomocą przekazywania komunikatów”).

## Przepływ danych z wykorzystaniem baz

W bazie proces zapisujący dane koduje je, a proces wczytujący dane z bazy je odkodowuje. Czasem z bazy korzysta tylko jeden proces. Wtedy proces odczytujący jest nowszą wersją samego siebie. W takiej sytuacji możesz traktować zapisywanie czegoś w bazie jak *przesyłanie komunikatu do swojej późniejszej wersji*.

Oczywiście niezbędne jest tu zachowanie zgodności wstecz. W przeciwnym razie późniejsza wersja nie będzie mogła odkodować tego, co zapisała wcześniejsza.

Zwykle w tym samym czasie z bazy korzysta kilka różnych procesów. Tymi procesami mogą być różne aplikacje lub usługi. Może to być także parę instancji tej samej usługi (działających równolegle w celu zapewnienia skalowalności lub odporności na błędy). Niezależnie od tego w środowisku, w którym aplikacje się zmieniają, prawdopodobnie niektóre procesy używające bazy będą obejmowały nowszy kod, a inne — starszy (np. z powodu instalowania nowej wersji w ramach stopniowej aktualizacji lub zaktualizowania tylko niektórych instancji).

To oznacza, że wartości w bazie mogą być zapisywane przez *nowszą* wersję kodu, a następnie wczytywane przez *starszą*, która wciąż jest używana. Dlatego w bazach często wymaga się zgodności wprzód.

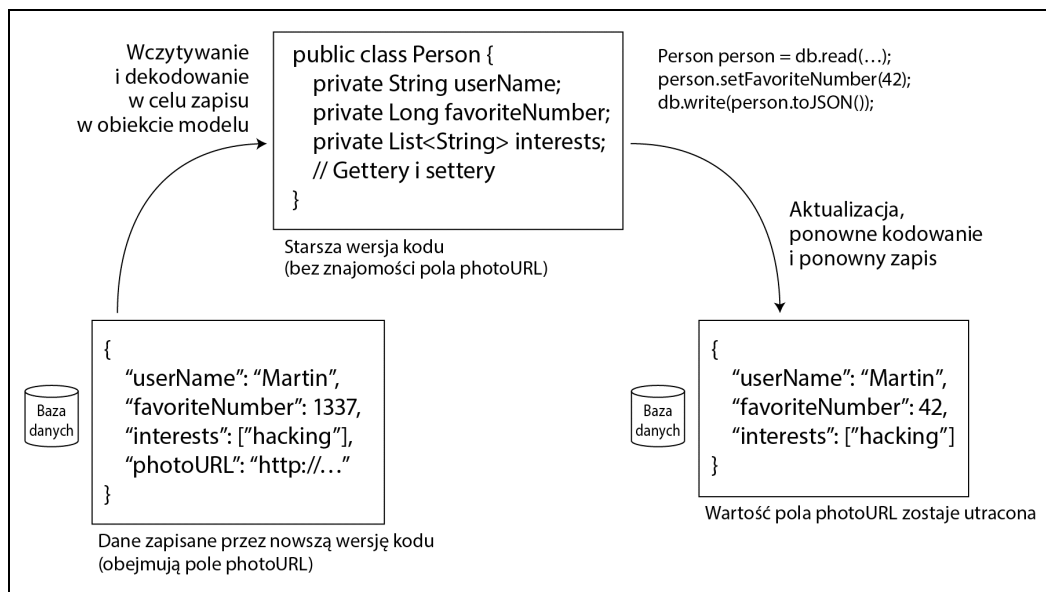
Występuje tu jednak dodatkowy problem. Załóżmy, że dodałeś pole do schematu rekordów, a nowszy kod zapisuje wartość tego nowego pola w bazie. Później starsza wersja kodu (która nie zna nowego pola) wczytuje ten rekord, aktualizuje go i ponownie zapisuje. W takiej sytuacji pożądanym rozwiązaniem jest zwykle zachowanie przez starszy kod nowego pola bez zmian, nawet gdy starszy kod nie potrafi tego pola zinterpretować.

Opisane wcześniej formaty kodowania obsługują zachowywanie nieznanych pól, jednak czasem trzeba zająć się tym problemem na poziomie aplikacji, co przedstawia rysunek 4.7. Na przykład jeśli dekodujesz wartość z bazy w obiektach modelu w aplikacji, a następnie ponownie kodujesz te obiekty, nieznane pole może zostać utracone w procesie przekształcania danych. Rozwiązanie tej kwestii nie jest trudne. Trzeba tylko o tym pamiętać.

### Różne wartości zapisywane w różnym czasie

Bazy zwykle umożliwiają aktualizowanie dowolnych wartości w dowolnym czasie. To oznacza, że w jednej bazie mogą się znajdować wartości zapisane pięć milisekund i pięć lat temu.

Gdy instalujesz nową wersję aplikacji, to — przynajmniej po stronie serwera — możesz całkowicie zastąpić starszą wersję nową w ciągu kilku minut. Z zawartością bazy jest inaczej. Dane sprzed pięciu lat zakodowane w pierwotny sposób nadal będą się w niej znajdować, chyba że w międzyczasie bezpośrednio je zastąpisz. To spostrzeżenie można podsumować w następujący sposób: *dane żyją dłużej niż kod*.



Rysunek 4.7. Jeśli nie zachowasz ostrożności, to gdy starsza wersja aplikacji aktualizuje dane zapisane wcześniej przez nowszą wersję, dane mogą zostać utracone

Ponowny zapis danych zgodnie z nowym schematem (*migracja*) jest oczywiście wykonalny. Jednak w przypadku dużych zbiorów danych to kosztowna operacja, dlatego w większości baz w miarę możliwości się jej unika. Większość baz relacyjnych pozwala na proste zmiany schematu (np. dodanie nowej kolumny o wartości domyślnej null) bez ponownego zapisu istniejących danych<sup>5</sup>. Gdy starszy wiersz jest wczytywany, baza dodaje wartość null we wszystkich kolumnach, których brakuje w zakodowanych danych z dysku. Dokumentowa baza danych Espresso w serwisie LinkedIn używa do składowania danych formatu Avro, dzięki czemu można wykorzystać w niej reguły ewolucji schematu z systemu Avro [23].

Tak więc dzięki ewolucji schematów cała baza może wyglądać tak, jakby była zakodowana przy użyciu jednego schematu, nawet jeśli w bazie znajdują się rekordy zakodowane za pomocą różnych historycznych wersji schematu.

## Składowanie danych archiwalnych

Możliwe, że od czasu do czasu zapisujesz snapshot bazy — np. w celu utworzenia kopii zapasowej lub wczytania danych do hurtowni (zob. punkt „Hurtownie danych”). Wtedy zrzucane dane są zwykle kodowane z użyciem najnowszego schematu, nawet jeśli dane w źródłowej bazie są zakodowane za pomocą różnych wersji schematu z różnych okresów. Ponieważ dane i tak są kopiowane, można przy okazji zakodować ich kopię w spójny sposób.

Z tego powodu że zrzucane dane są zapisywane w jednej operacji, po czym stają się niezmiennie, dobrze sprawdzają się tu formaty takie jak obiektowe pliki kontenerowe Avro. To też dobra oka-

<sup>5</sup> Wyjątkiem jest baza MySQL, gdzie często cała tabela jest zapisywana ponownie nawet wtedy, gdy nie jest to bezwzględnie konieczne (zob. punkt „Elastyczność schematu w modelu opartym na dokumentach”).

zja do zakodowania danych w wygodnym do analityki formacie kolumnowym takim jak Parquet (zob. punkt „Kompresja kolumn”).

W rozdziale 10. dowiesz się więcej na temat używania danych z baz archiwalnych.

## Przepływ danych z użyciem usług: REST i RPC

Gdy procesy muszą komunikować się przez sieć, istnieje kilka różnych sposobów zarządzania taką komunikacją. Najczęściej spotykane rozwiązanie to określenie dwóch ról: *klientów* i *serwerów*. Serwery udostępniają interfejs API przez sieć, a klienci mogą się łączyć z serwerami, kierując żądania do tego interfejsu API. Interfejs API udostępniany przez serwer to *usługa*.

Internet działa w następujący sposób: klienci (przeglądarki internetowe) kierują żądania do serwerów WWW, zgłaszając żądania GET (w celu pobierania kodu w HTML-u, CSS-ie i JavaScriptcie, grafiki itd.) i POST (na potrzeby przesyłania danych na serwer). Interfejs API obsługuje ustandaryzowany zestaw protokołów i formatów danych (HTTP, URL, SSL/TLS, HTML itd.). Ponieważ twórcy przeglądarek, serwerów WWW i witryn w większości przestrzegają tych standardów, możesz się posłużyć dowolną przeglądarką, aby wyświetlić dowolną witrynę (przynajmniej teoretycznie!).

Przeglądarki nie są jedynym typem klientów. Na przykład aplikacje natywne działające w urządzeniach przenośnych lub komputerach stacjonarnych także mogą kierować żądania sieciowe do serwera, a aplikacje w JavaScriptcie działające w przeglądarce po stronie klienta mogą się posługiwać żadaniami XMLHttpRequest, aby działać jako klient HTTP (ten model to *Ajax* [30]). W takiej sytuacji odpowiedzią z serwera zwykle nie jest kod w HTML-u przeznaczony do wyświetlania ludziom, ale dane zakodowane w sposób wygodny do dalszego przetwarzania przez kod aplikacji po stronie klienta (np. w formacie JSON). Choć protokołem transportowym może być wtedy HTTP, zaimplementowany na jego bazie interfejs API jest specyficzny dla aplikacji, a klient i serwer muszą uzgodnić szczegóły tego interfejsu.

Ponadto sam serwer może być klientem innej usługi (np. typowy serwer aplikacji sieciowej to klient bazy danych). To podejście jest często stosowane w celu podziału dużej aplikacji na mniejsze usługi według funkcji. Na przykład jedna usługa może zgłaszać żądania do innej, gdy potrzebuje od niej jakiejś funkcji lub danych. Ten sposób budowania aplikacji tradycyjnie był nazywany **SOA** (ang. *service-oriented architecture*), jednak ostatnio przemianowano go na *architekturę mikrousług* [31, 32].

Usługi są pod niektórymi względami podobne do baz. Zwykle umożliwiają klientom przekazywanie danych i zgłaszanie zapytań o nie. Jednak bazy pozwalają na zgłaszanie arbitralnych zapytań z użyciem opisanych w rozdziale 2. języków zapytań, natomiast usługi udostępniają specyficzny dla aplikacji interfejs API, który obsługuje wyłącznie wejścia i wyjścia wstępnie określone w logice biznesowej (kodzie aplikacji) usługi [33]. To ograniczenie zapewnia pewien poziom hermetyzacji — usługi mogą narzucać precyzyjne ograniczenia na to, co jest dozwolone klientom.

Ważny cel projektowy w architekturze mikrousług (SOA) to ułatwienie modyfikowania i konserwacji aplikacji dzięki temu, że usługi można instalować i zmieniać niezależnie od pozostałych. Każdą usługą może się opiekować jeden zespół, który powinien móc często udostępniać nowe wersje tej usługi bez konieczności koordynowania prac z innymi zespołami. Oznacza to, że należy

oczekiwać, iż w tym samym czasie będą działać starsze i nowsze wersje serwerów i klientów. Dlatego kodowanie danych używane przez serwery i klienty musi być zgodne w poszczególnych wersjach interfejsu API usługi. Właśnie tej kwestii poświęcono ten rozdział.

## Usługi internetowe

Gdy do komunikowania się z usługą używa się protokołu HTTP, jest ona nazywana *usługą internetową*. Nazwa ta może być nieco myląca, ponieważ usługi internetowe są używane nie tylko w internecie, ale w różnych kontekstach. Oto przykłady:

1. Aplikacja kliencka działająca w urządzeniu użytkownika (np. aplikacja natywna w urządzeniu przenośnym lub aplikacja sieciowa w JavaScriptcie używająca Ajaksa) zgłasza żądanie do usługi za pomocą protokołu HTTP. Takie żądania zwykle są przesyłane przez publiczny internet.
2. Jedna usługa zgłasza żądanie do innej usługi należącej do tej samej firmy i często znajdującej się w tym samym centrum danych. Usługi te działają w ramach architektury mikrousług (SOA). Oprogramowanie stosowane do obsługi takich scenariuszy bywa czasem nazywane *warstwą pośrednią*.
3. Jedna usługa zgłasza żądania do usługi należącej do innej firmy, zwykle wykorzystując internet. To rozwiązanie jest używane do wymiany danych między systemami zaplecza z różnych organizacji. Do tej kategorii należą publiczne interfejsy API udostępniane przez usługi elektroniczne (np. systemy obsługi kart kredytowych) lub systemy korzystające ze standardu OAuth na potrzeby wspólnego dostępu do danych użytkownika.

Istnieją dwa popularne modele tworzenia usług internetowych: *REST* i *SOAP*. Są one prawie swoim przeciwieństwem, jeśli chodzi o filozofię, a ich zwolennicy często toczą zacięte dyskusje<sup>6</sup>.

REST nie jest protokołem, a raczej filozofią projektową opartą na zasadach protokołu HTTP [34, 35]. Nacisk położono tu na proste formaty danych, używanie adresów URL do identyfikowania zasobów i używanie mechanizmów protokołu HTTP do zarządzania pamięcią podręczną, uwierzytelniania i uzgadniania typu treści. REST w porównaniu z SOAP zyskał popularność (przynajmniej w kontekście integrowania usług między organizacjami [36]) i często jest łączony z mikrousługami [31]. Interfejs API zaprojektowany zgodnie z zasadami podejścia REST nosi nazwę *RESTful*.

Natomiast SOAP to protokół oparty na XML-u służący do zgłaszania w sieci żądań do interfejsów API<sup>7</sup>. Choć najczęściej wykorzystuje protokół HTTP, ma być niezależny od niego i unikać większości mechanizmów HTTP. Jest on dość rozwlekły i wykorzystuje skomplikowany zestaw powiązanych standardów (to *platforma usług internetowych* nazywana *WS-\**), które zapewniają różne funkcje [37].

Interfejs API usługi internetowej SOAP został opisany w opartym na XML-u języku **WSDL** (ang. *Web Services Description Language*). WSDL umożliwia generowanie kodu, dzięki czemu klienty mogą uzyskać dostęp do zdalnej usługi za pomocą lokalnych klas i wywołań metod (które są ko-

<sup>6</sup> Nawet w ramach każdego z obozów toczono są liczne dyskusje. Ich tematem często jest np. **HATEOAS** (ang. *hypermedia as the engine of application state*).

<sup>7</sup> Mimo podobieństwa akronimów SOAP nie jest konieczny do tworzenia architektury SOA. SOAP to konkretna technologia, natomiast SOA to ogólne podejście do budowania systemów.

dowane jako komunikaty w XML-u i później dekodowane przez platformę). To podejście przydaje się w językach programowania ze statycznym określaniem typów (w językach z dynamicznym określaniem typów jest ono mniej użyteczne; zob. punkt „Generowanie kodu i języki z dynamicznym określaniem typów”).

Ponieważ WSDL nie został zaprojektowany jako czytelny dla człowieka, a komunikaty SOAP są często zbyt skomplikowane, by tworzyć je ręcznie, użytkownicy protokołu SOAP w dużym stopniu polegają na pomocy narzędzi, generowaniu kodu i środowiskach IDE [38]. Dla użytkowników języków programowania nieobsługiwanych przez producentów narzędzi dla protokołu SOAP integracja z usługami SOAP jest trudna.

Choć SOAP i jego różne rozszerzenia są pozornie ustandaryzowane, współdziałanie między implementacjami różnych producentów często rodzi problemy [39]. Wszystkie te powody sprawiają, że choć w wielu dużych firmach SOAP nadal jest używany, większość mniejszych organizacji zrezygnowała z tego protokołu.

W interfejsach API typu RESTful zwykle faworyzuje się prostsze rozwiązania, zazwyczaj obejmujące mniej generowania kodu i zautomatyzowanych narzędzi. Do opisu interfejsów API typu RESTful i tworzenia dokumentacji można się posłużyć formatem definicji takim jak OpenAPI (jego inna nazwa to Swagger [40]).

## Problemy z wywołaniami RPC

Usługi internetowe to najnowsza odmiana długiej linii technologii do zgłaszania żądań API w sieci. Wiele tych technologii zyskało dużą popularność, jednak wiąże się z nimi wiele problemów. Enterprise JavaBeans (EJB) i Remote Method Invocation (RMI) działają tylko w Javie. Distributed Component Object Model (DCOM) jest obsługiwana tylko w systemach Microsoftu. Common Object Request Broker Architecture (CORBA) jest nadmiernie złożona i nie zapewnia zgodności wstecz ani wprzód [41].

Wszystkie wymienione technologie są oparte na *wywołaniach RPC* (ang. *remote procedure call*) stosowanych już od lat 70. [42]. W modelu RPC żądania kierowane do zdalnej usługi sieciowej mają wyglądać tak samo jak wywołania funkcji lub metod w danym języku programowania i w tym samym procesie (tę abstrakcję nazywa się *przezroczystością lokalizacji*). Choć wywołania RPC na pozór wydają się wygodnym rozwiązaniem, to podejście to ma zasadnicze wady [43, 44]. Żądania sieciowe wyraźnie różnią się od lokalnych wywołań funkcji:

- Wywołania lokalnej funkcji są przewidywalne i zawsze się kończą powodzeniem lub niepowodzeniem, na co wpływają tylko parametry, nad którymi zachowujesz kontrolę. Żądanie sieciowe jest nieprzewidywalne. Żądanie lub odpowiedź mogą zostać utracone z powodu problemów w sieci, a zdalna maszyna może być powolna lub niedostępna. Takie problemy pozostają całkowicie poza Twoją kontrolą. Problemy z siecią są częste, dlatego trzeba ich oczekiwać — np. ponawiając próby po żądaniach zakończonych niepowodzeniem.
- Wywołanie lokalnej funkcji albo zwraca wynik, albo zgłasza wyjątek, albo nigdy nie zwraca sterowania (z powodu uruchomienia pętli nieskończonej lub wystąpienia awarii procesu). W żądaniu sieciowym możliwy jest inny efekt. Może ono zwrócić sterowanie bez wyniku z powodu *przekroczenia limitu czasu*. Nie wiadomo wtedy, co się stało. Jeśli nie otrzymałeś odpowiedzi od zdalnej usługi, nie możesz stwierdzić, czy żądanie do niej dotarło, czy nie. Ten temat został omówiony szczegółowo w rozdziale 8.

- Jeśli ponawiasz żądanie sieciowe zakończone niepowodzeniem, możliwe, że żądanie dociera do celu, a tylko odpowiedzi zostają utracone. Wtedy ponawianie próby skutkuje wielokrotnym wykonaniem danej operacji — chyba że wbudujesz w protokół mechanizm deduplikacji (zapewniania *idempotencji*). W lokalnych wywołaniach funkcji ten problem nie występuje. Szczegółowe omówienie idempotencji zawiera rozdział 11.
- Obsługa każdego wywołania lokalnej funkcji zwykle zajmuje mniej więcej tyle samo czasu. Żądania sieciowe są znacznie wolniejsze niż wywołania funkcji, a czas ich przetwarzania znacznie się waha. W dobrym momencie żądanie może zostać obsłużone w czasie krótszym niż milisekunda. Jednak gdy sieć lub zdalna usługa jest przeciążona, wykonanie tej samej operacji może zająć wiele sekund.
- Gdy wywołujesz funkcję lokalną, możesz wydajnie przekazać do niej referencje (wskaźniki) do obiektów z pamięci lokalnej. W żądaniach sieciowych wszystkie parametry trzeba zakodować do postaci sekwencji bajtów, którą można przesłać w sieci. Nie stanowi to problemu, gdy parametrami są wartości typów prostych (np. liczby lub łańcuchy znaków), jednak w przypadku większych obiektów pojawiają się trudności.
- Klient i usługa mogą być zaimplementowane w różnych językach programowania. Dlatego platforma wywołań RPC musi przekształcać typy danych z jednego języka na inny. Może to być kłopotliwe, ponieważ nie wszystkie języki udostępniają te same typy danych. Przypomnij sobie np. problemy związane z wartościami większymi niż  $2^{33}$  w JavaScriptcie (zob. punkt „JSON, XML i ich wersje binarne”). W jednym procesie napisanym w jednym języku takie trudności nie występują.

Wszystkie wymienione czynniki oznaczają, że nie mają sensu próby upodobnienia zdalnej usługi do lokalnych obiektów z danego języka programowania, ponieważ są to z natury różne rzeczy. Jeden z atrakcyjnych aspektów podejścia REST to fakt, że nie próbuje ono ukryć faktu, iż jest protokołem sieciowym (choć nie powstrzymuje to niektórych osób od budowania bibliotek do obsługi wywołań RPC na bazie protokołu REST).

## Obecne kierunki rozwoju wywołań RPC

Mimo wymienionych problemów wywołania RPC pozostaną w użyciu. Na podstawie wszystkich kodowań przedstawionych w tym rozdziale powstały różne platformy RPC. Na przykład Thrift i Avro obejmują wbudowaną obsługę wywołań RPC, gRPC to implementacja wywołań RPC używająca formatu Protocol Buffers, Finagle także używa Thrifta, a w Rest.li używane są format JSON i protokół HTTP.

W tej nowej generacji platform RPC bardziej bezpośrednio przyjmuje się, że zdalne żądanie różni się od wywołania funkcji lokalnej. Na przykład w Finagle i Rest.li stosuje się *obietnice* (ang. *futures* lub *promises*) pozwalające ukryć asynchroniczne operacje, które mogą się zakończyć niepowodzeniem. Obietnice upraszczają też pracę w sytuacjach, gdy musisz równolegle zgłaszać żądania do wielu usług i łączyć wyniki [45]. Platforma gRPC obsługuje *strumienie*, w których wywołanie zamiast jednego żądania i jednej odpowiedzi obejmuje serię żądań i odpowiedzi [46].



Niektóre z wymienionych platform obsługują *wykrywanie usług*, czyli umożliwiają klientowi ustalenie, pod jakim adresem IP i numerem portu mogą znaleźć konkretną usługę. Ten temat opisano w punkcie „Trasowanie żądań”.

Niestandardowe protokoły RPC z kodowaniem binarnym pozwalają uzyskać wyższą wydajność niż uniwersalne rozwiązania takie jak JSON w połączeniu z REST. Jednak interfejsy API typu RESTful mają inne istotne zalety: dobrze się nadają do eksperymentowania i debugowania (możesz przesyłać żądanie do takiego interfejsu za pomocą przeglądarki lub działającego w wierszu poleceń narzędzia curl, co nie wymaga generowania kodu ani instalowania oprogramowania), są obsługiwane we wszystkich popularnych językach programowania i platformach, a ponadto dostępny jest dla nich rozbudowany ekosystem narzędzi (serwerów, pamięci podręcznych, równoważników obciążenia, serwerów pośredniczących, zapór, systemów monitorowania, narzędzi diagnostycznych i testowych itd.).

Z tych powodów REST wydaje się dominującym sposobem tworzenia publicznych interfejsów API. W platformach RPC nacisk położono głównie na żądania przesyłane między usługami należącymi do tej samej organizacji i zwykle działającymi w tym samym centrum danych.

### Kodowanie danych i modyfikacje w modelu RPC

Jeśli chodzi o łatwość modyfikowania, ważne jest, by klienty i serwery RPC można było modyfikować i instalować niezależnie od siebie. W porównaniu z przepływem danych przez bazy (co opisano we wcześniejszym punkcie) w kontekście przepływu danych przez usługi można przyjąć upraszczające założenie dotyczące tego, że najpierw aktualizowane są wszystkie serwery, a dopiero potem klienty. Dlatego trzeba tylko zachować zgodność wstecz dla żądań i zgodność wprzód dla odpowiedzi.

Zgodność wstecz i wprzód schematu używanego w przypadku wywołań RPC wynika z zastosowanego kodowania:

- Wywołania RPC w systemach Thrift, gRPC (Protocol Buffers) i Avro umożliwiają modyfikacje zgodnie z regułami zapewniania zgodności z używanego formatu kodowania.
- W protokole SOAP specyfikacje żądań i odpowiedzi mają postać schematów XML-owych. Można je modyfikować, jednak występują w tym obszarze subtelne pułapki [47].
- W interfejsach API typu RESTful dla odpowiedzi najczęściej używa się formatu JSON (bez formalnie określonego schematu), a dla żądań stosuje się parametry w formacie JSON albo zakodowane w identyfikatorze URI lub danych z formularza. Zwykle uważa się, że zmiany polegające na dodawaniu opcjonalnych parametrów żądań i nowych pól do obiektów z odpowiedzią nie naruszają zgodności.

Zachowanie zgodności usług jest utrudnione przez to, że wywołania RPC często są stosowane do komunikowania się między organizacjami. Dlatego dostawca usługi nieraz nie ma kontroli nad klientami i nie może wymusić ich aktualizacji. Stąd trzeba zachowywać zgodność przez długi czas (możliwe, że w nieskończoność). Jeśli konieczna jest zmiana naruszająca zgodność, dostawca usługi często równolegle utrzymuje wiele wersji interfejsu API usługi.

Nie ma zgody co do tego, jak powinna działać kontrola wersji interfejsów API (związana z tym, jak klient ma określić, którą wersję interfejsu API chce zastosować [48]). W interfejsach API typu RESTful popularne podejście to stosowanie numeru wersji w adresie URL lub nagłówku HTTP

Accept. W usługach używających do identyfikowania konkretnego klienta kluczy interfejsu API inną możliwością stanowi zapisanie na serwerze wersji interfejsu API żądanej przez danego klienta i umożliwienie aktualizacji wybranej wersji za pomocą odrębnego interfejsu administracyjnego [49].

## Przepływ danych za pomocą przekazywania komunikatów

Opisano już różne sposoby przepływu zakodowanych danych z jednego procesu do innego. Do tej pory omówiono modele REST i RPC (w których jeden proces przesyła żądanie przez sieć do innego procesu i oczekuje jak najszybszego odesłania odpowiedzi) oraz bazy danych (gdzie jeden proces zapisuje zakodowane dane, a inny wczytuje je ponownie w przyszłości).

W tym końcowym punkcie pokrótce opisane są systemy *asynchronicznego przesyłania komunikatów*. To rozwiązanie pośrednie między wywołaniami RPC a bazami danych, podobne do wywołań RPC pod tym względem, że żądanie klienta (nazywane zwykle *komunikatem*) jest dostarczane do innego procesu z małym opóźnieniem. Podobieństwo do baz danych polega na tym, że komunikat nie jest przesyłany zwykłym połączeniem sieciowym, ale przez pośrednika nazywanego *brokerem komunikatów* (inne określenia to *kolejka komunikatów* lub *warstwa pośrednia komunikatów*), który tymczasowo przechowuje komunikat.

Posługiwanie się brokerem komunikatów ma kilka zalet w porównaniu do bezpośrednich wywołań RPC:

- Broker może działać jak bufor, jeśli odbiorca jest niedostępny lub przeciążony. Zwiększa to niezawodność systemu.
- Broker może automatycznie ponownie przysyłać komunikaty do procesu, który uległ awarii. Zapobiega to utracie komunikatów.
- Nadawca nie musi znać adresu IP i numeru portu odbiorcy (przydaje się to zwłaszcza w chmurze, gdzie maszyny wirtualne często pojawiają się i znikają).
- Można przesłać jeden komunikat do wielu odbiorców.
- Nadawca jest logicznie oddzielony od odbiorcy (nadawca jedynie publikuje komunikaty i nie ma dla niego znaczenia, kto je odbierze).

Jednak różnicą w porównaniu z wywołaniami RPC jest to, że przekazywanie komunikatów odbywa się zwykle jednokierunkowo. Nadawca zwykle nie oczekuje, że otrzyma odpowiedź na komunikaty. Możliwe, że proces prześle odpowiedź, jednak przeważnie odbywa się to w osobnym kanale. To *asynchroniczny wzorzec komunikacji*. Nadawca nie czeka na dostarczenie komunikatu, ale po prostu przesyła go i o nim zapomina.

### Brokery komunikatów

W przeszłości rynek brokerów komunikatów był zdominowany przez komercyjne oprogramowanie dla przedsiębiorstw rozwijane przez firmy takie jak TIBCO, IBM WebSphere i webMethods. Później popularność zyskały otwarte implementacje, np. RabbitMQ, ActiveMQ, HornetQ, NATS i Apache Kafka. Ich porównanie zawiera rozdział 11.

Szczegółowa semantyka dostarczania komunikatów zależy od implementacji i konfiguracji. Ogólnie brokery komunikatów są używane w następujący sposób: jeden proces przesyła komunikat

do *kolejki* lub *tematu* o określonej nazwie, a broker gwarantuje, że ten komunikat zostanie dostarczony do jednego lub kilku *konsumentów* lub *subskrybentów* kolejki lub tematu. Z tym samym tematem może być powiązanych wielu producentów i konsumentów.

Temat umożliwia przepływ danych tylko w jednym kierunku. Jednak konsument może publikować komunikaty w innym temacie (co pozwala połączyć je w łańcuch, jak to opisano w rozdziale 11.) lub w kolejce odpowiedzi, z której korzysta nadawca pierwotnego komunikatu, co umożliwia przepływ danych w modelu żądanie – odpowiedź (podobnie jak w wywołaniach RPC).

Brokery komunikatów zwykle nie wymuszają stosowania żadnego konkretnego modelu danych. Komunikat to tylko sekwencja bajtów z metadanymi, dzięki czemu można się posługiwać dowolnym formatem kodowania. Jeśli kodowanie jest zgodne wstecz i wprzód, opisane podejście zapewnia największą elastyczność w zakresie niezależnej zmiany nadawców i konsumentów oraz instalowania ich w dowolnej kolejności.

Jeżeli konsument ponownie publikuje komunikaty w innym temacie, należy postępować ostrożnie, aby zachować nieznane pola w celu uniknięcia problemu opisanego wcześniej w kontekście baz danych (rysunek 4.7).

## Platformy z rozproszonymi aktorami

*Model aktora* to model programowania umożliwiający uzyskanie współbieżności w jednym procesie. Zamiast bezpośrednio zmagać się z wątkami (i powiązаныmi problemami warunku wyścigu, blokad i zakleszczeń), należy umieścić logikę w *aktorach*. Każdy aktor reprezentuje zwykle jednego klienta lub jednostkę i może mieć stan lokalny (który nie jest współużytkowany z innymi aktorami), a komunikuje się z innymi aktorami przez wysyłanie i odbieranie asynchronicznych komunikatów. Dostarczanie komunikatów nie jest gwarantowane. W niektórych scenariuszach błędy powodują, że komunikaty zostaną utracone. Ponieważ każdy aktor przetwarza w danym momencie tylko jeden komunikat, nie trzeba uwzględniać wątków. Ponadto platforma może niezależnie planować pracę każdego aktora.

W *platformach z rozproszonymi aktorami* tego modelu programowania używa się do skalowania aplikacji z użyciem wielu węzłów. Ten sam mechanizm przekazywania komunikatów jest stosowany niezależnie od tego, czy nadawca i odbiorca działają w jednym węźle, czy w różnych węzłach. Jeśli komponenty działają w różnych węzłach, komunikat jest automatycznie kodowany jako sekwencja bajtów, przesyłany przez sieć i dekodowany po drugiej stronie.

Przezroczystość lokalizacji działa lepiej w modelu aktorów niż w wywołaniach RPC, ponieważ w modelu aktorów od początku przyjmuje się, że komunikaty mogą zostać utracone (nawet w jednym procesie). Choć opóźnienie w sieci jest zwykle większe niż w jednym procesie, w modelu aktorów występują mniejsze fundamentalne różnice między komunikacją lokalną a zdalną.

Platforma z rozproszonymi aktorami integruje brokera komunikatów i model programowania z użyciem aktorów w jedną platformę. Jeśli jednak chcesz przeprowadzić stopniową aktualizację aplikacji opartej na aktorach, musisz zadbać o zgodność wprzód i wstecz, ponieważ komunikaty mogą być przesyłane z węzła używającego nowej wersji do węzła używającego starszej wersji i na odwrót.

Poniżej opisano, jak trzy popularne platformy z rozproszonymi aktorami obsługują kodowanie komunikatów:

- *Akka* domyślnie używa wbudowanej serializacji z Javy, która nie zapewnia zgodności wprzód ani wstecz. Można ją jednak zastąpić np. mechanizmem Protocol Buffers i zyskać w ten sposób możliwość przeprowadzania stopniowych aktualizacji [50].
- *Orleans* domyślnie stosuje niestandardowy format kodowania danych, który nie umożliwia stopniowej aktualizacji. Aby zainstalować nową wersję aplikacji, trzeba skonfigurować nowy klastrer, przenieść ruch ze starego klastra do nowego, a następnie wyłączyć stary [51, 52]. Podobnie jak w Akce można stosować niestandardowe wtyczki do obsługi serializacji.
- W platformie *Erlang OTP* zaskakująco trudno jest wprowadzać zmiany w schematach rekordów (choć system ma wiele mechanizmów zaprojektowanych z myślą o wysokiej dostępności). Stopniowe aktualizacje są możliwe, choć trzeba je starannie zaplanować [53]. W przyszłości może to ułatwić nowy eksperymentalny typ danych maps (to wprowadzona w 2014 r. w Erlangu R17 struktura zbliżona do JSON-a) [54].

## Podsumowanie

W tym rozdziale opisano kilka sposobów przekształcania struktur danych w bajty w sieci lub na dysku. Pokazano, że szczegóły kodowań wpływają nie tylko na ich wydajność, ale też — co ważniejsze — na architekturę aplikacji i możliwości ich wdrażania.

Wiele usług wymaga obsługi stopniowej aktualizacji, kiedy to nowa wersja usługi jest stopniowo wdrażana w grupach węzłów, a nie we wszystkich węzłach jednocześnie. Stopniowe aktualizacje umożliwiają udostępnianie nowych wersji usługi bez przestojów (co zachęca do częstszego udostępniania wersji z niewielkimi zmianami zamiast do rzadkiego udostępniania wielu zmian) i zmniejszają ryzyko wdrożeń (ponieważ można wykryć i wycofać błędne wersje, zanim problem dotknie wielu użytkowników). Te cechy bardzo korzystnie wpływają na *łatwość modyfikowania*, czyli na prostotę wprowadzania zmian w aplikacji.

W trakcie stopniowej aktualizacji lub z różnych innych przyczyn trzeba zakładać, że różne węzły korzystają z różnych wersji kodu aplikacji. Dlatego ważne jest, aby wszystkie dane przepływające w systemie były zakodowane w sposób zapewniający zgodność wstecz (nowy kod może czytać starsze dane) i wprzód (starszy kod może czytać nowe dane).

Omówiono tu kilka formatów kodowania danych i ich cechy związane ze zgodnością:

- Kodowania specyficzne dla języka programowania są ograniczone do jednego języka i często nie zapewniają zgodności wprzód ani wstecz.
- Formaty tekstowe takie jak JSON, XML i CSV są powszechnie dostępne, a zachowanie zgodności zależy tu od tego, jak są używane. Dostępne są opcjonalne języki schematów, które czasem bywają pomocne, a czasem okazują się przeszkodą. W tych formatach typy danych nie są precyzyjnie określone, dlatego trzeba zachować ostrożność np. w przypadku liczb i binarnych łańcuchów znaków.

- Formaty binarne oparte na schematach (np. Thrift, Protocol Buffers i Avro) umożliwiają zwarte, wydajne kodowanie z jasno zdefiniowaną semantyką zachowywania zgodności wprzód i wstecz. Te schematy mogą być przydatne do generowania dokumentacji i kodu w językach ze statycznie określonymi typami. Mają jednak tę wadę, że dane trzeba zdekodować, aby były czytelne dla człowieka.

Opisano też kilka sposobów przepływu danych, ilustrując różne scenariusze, w których kodowania danych mają znaczenie:

- Bazy, gdzie proces zapisujący dane w bazie koduje je, a proces wczytujący dane z bazy je dekoduje.
- Interfejsy API typu RPC i REST, gdzie klient koduje żądanie, serwer dekoduje je i koduje odpowiedź, a na końcu klient dekoduje odpowiedź.
- Asynchroniczne przekazywanie komunikatów (z użyciem brokerów komunikatów lub aktorów), gdzie węzły się komunikują, przysyłając między sobą komunikaty kodowane przez nadawcę i dekodowane przez odbiorcę.

W podsumowaniu można napisać, że zachowując nieco ostrożności, da się uzyskać zgodność wstecz i wprzód oraz umożliwić stopniowe aktualizacje. Niech ewolucja Twojej aplikacji będzie błyskawiczna, a wdrożenia częste!

## Literatura cytowana

- [1] *Java Object Serialization Specification*, docs.oracle.com, 2010 (<https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html>).
- [2] *Ruby 2.2.0 API Documentation*, ruby-doc.org, grudzień 2014 (<http://ruby-doc.org/core-2.2.0/>).
- [3] *The Python 3.4.3 Standard Library Reference Manual*, docs.python.org, luty 2015 (<https://docs.python.org/3/library/pickle.html>).
- [4] *EsotericSoftware/kryo*, github.com, październik 2014 (<https://github.com/EsotericSoftware/kryo>).
- [5] *CWE-502: Deserialization of Untrusted Data*, Common Weakness Enumeration, cwe.mitre.org, 30 lipca 2014 (<http://cwe.mitre.org/data/definitions/502.html>).
- [6] Steve Breen, *What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability*, foxglovesecurity.com, 6 listopada 2015 (<https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/>).
- [7] Patrick McKenzie, *What the Rails Security Issue Means for Your Startup*, kalzumeus.com, 31 stycznia 2013 (<http://www.kalzumeus.com/2013/01/31/what-the-rails-security-issue-means-for-your-startup/>).
- [8] Eishay Smith, *jvm-serializers wiki*, github.com, listopad 2014 (<https://github.com/eishay/jvm-serializers/wiki>).
- [9] *XML Is a Poor Copy of S-Expressions*, wiki c2.com (<http://wiki.c2.com/?XmlIsAPoorCopyOfEssExpressions>).

- [10] Matt Harris, *Snowflake: An Update and Some Very Important Information*, e-mail na liście dyskusyjnej *Twitter Development Talk*, 19 października 2010 (<https://groups.google.com/forum/#!topic/twitter-development-talk/ahbvo3VTIYI>).
- [11] Shudi (Sandy) Gao, C. Michael Sperberg-McQueen i Henry S. Thompson, *XML Schema 1.1, W3C Recommendation*, maj 2001 (<https://www.w3.org/XML/Schema>).
- [12] Francis Galiegue, Kris Zyp i Gary Court, *JSON Schema*, IETF InternetDraft, luty 2013 (<http://json-schema.org/>).
- [13] Yakov Shafranovich, *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*, październik 2005 (<https://tools.ietf.org/html/rfc4180>).
- [14] *MessagePack Specification*, msgpack.org (<https://msgpack.org/>).
- [15] Mark Slee, Aditya Agarwal i Marc Kwiatkowski, *Thrift: Scalable Cross-Language Services Implementation*, raport techniczny Facebooka, kwiecień 2007 (<http://thrift.apache.org/static/files/thrift-20070401.pdf>).
- [16] *Protocol Buffers Developer Guide*, Google, Inc., developers.google.com (<https://developers.google.com/protocol-buffers/docs/overview>).
- [17] Igor Anishchenko, *Thrift vs Protocol Buffers vs Avro - Biased Comparison*, slideshare.net, 17 września 2012 (<https://www.slideshare.net/IgorAnishchenko/pb-vs-thrift-vs-avro>).
- [18] *A Matrix of the Features Each Individual Language Library Supports*, wiki.apache.org (<https://wiki.apache.org/thrift/LibraryFeatures>).
- [19] Martin Kleppmann, *Schema Evolution in Avro, Protocol Buffers and Thrift*, martin.kleppmann.com, 5 grudnia 2012 (<http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html>).
- [20] *Apache Avro 1.7.7 Documentation*, avro.apache.org, lipiec 2014 (<http://avro.apache.org/docs/1.7.7/>).
- [21] Doug Cutting, Chad Walters, Jim Kellerman i in., *[PROPOSAL] New Subproject: Avro*, wątek na liście dyskusyjnej *hadoop-general*, mail-archives.apache.org, kwiecień 2009 ([http://mail-archives.apache.org/mod\\_mbox/hadoop-general/200904.mbox/%3C49D53694.1050906@apache.org%3E](http://mail-archives.apache.org/mod_mbox/hadoop-general/200904.mbox/%3C49D53694.1050906@apache.org%3E)).
- [22] Tony Hoare, *Null References: The Billion Dollar Mistake*, w: „QCon London”, marzec 2009 (<https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>).
- [23] Aditya Auradkar i Tom Quiggle, *Introducing Espresso — LinkedIn’s Hot New Distributed Document Store*, engineering.linkedin.com, 21 stycznia 2015 (<https://engineering.linkedin.com/espresso/introducing-espresso-linkedins-hot-new-distributed-document-store>).
- [24] Jay Kreps, *Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2)*, blog.confluent.io, 25 lutego 2015 (<https://www.confluent.io/blog/stream-data-platform-2/>).

- [25] Gwen Shapira, *The Problem of Managing Schemas*, [radar.oreilly.com](http://radar.oreilly.com/2014/11/the-problem-of-managing-schemas.html), 4 listopada 2014 (<http://radar.oreilly.com/2014/11/the-problem-of-managing-schemas.html>).
- [26] *Apache Pig 0.14.0 Documentation*, [pig.apache.org](http://pig.apache.org/docs/r0.14.0/), listopad 2014 (<http://pig.apache.org/docs/r0.14.0/>).
- [27] John Larmouth, *ASN.1 Complete*, Morgan Kaufmann, 1999, ISBN: 978-0-122-33435-1 (<http://www.oss.com/asn1/resources/books-whitepapers-pubs/larmouth-asn1-book.pdf>).
- [28] Russell Housley, Warwick Ford, Tim Polk i David Solo, *RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile*, IETF Network Working Group, Standards Track, styczeń 1999 (<https://www.ietf.org/rfc/rfc2459.txt>).
- [29] Lev Walkin, *Question: Extensibility and Dropping Fields*, [lionet.info](http://lionet.info), 21 września 2010 (<http://lionet.info/asn1c/blog/2010/09/21/question-extensibility-removing-fields/>).
- [30] Jesse James Garrett, *Ajax: A New Approach to Web Applications*, [adaptivepath.com](http://adaptivepath.com), 18 lutego 2005 (<http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>).
- [31] Sam Newman, *Building Microservices*, O'Reilly Media, 2015, ISBN: 978-1-491-95035-7.
- [32] Chris Richardson, *Microservices: Decomposing Applications for Deployability and Scalability*, [infoq.com](http://infoq.com), 25 maja 2014 (<https://www.infoq.com/articles/microservices-intro>).
- [33] Pat Helland, *Data on the Outside Versus Data on the Inside*, w: „2nd Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2005 (<http://cidrdb.org/cidr2005/papers/P12.pdf>).
- [34] Roy Thomas Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, rozprawa doktorska, University of California, Irvine, 2000 ([https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)).
- [35] Roy Thomas Fielding, *REST APIs Must Be Hypertext-Driven*, [roy.gbiv.com](http://roy.gbiv.com), 20 października 2008 (<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>).
- [36] *REST in Peace, SOAP*, [royal.pingdom.com](http://royal.pingdom.com), 15 października 2010 (<http://royal.pingdom.com/2010/10/15/rest-in-peace-soap/>).
- [37] *Web Services Standards as of Q1 2007*, [innoc.com](http://www.innoq.com), luty 2007 (<https://www.innoq.com/resources/ws-standards-poster/>).
- [38] Pete Lacey, *The S Stands for Simple*, [harmful.cat-v.org](http://harmful.cat-v.org), 15 listopada 2006 (<http://harmful.cat-v.org/software/xml/soap/simple>).
- [39] Stefan Tilkov, *Interview: Pete Lacey Criticizes Web Services*, [infoq.com](http://www.infoq.com), 12 grudnia 2006 (<https://www.infoq.com/articles/pete-lacey-ws-criticism>).
- [40] *OpenAPI Specification (fka Swagger RESTful API Documentation Specification) Version 2.0*, [swagger.io](http://swagger.io), 8 września 2014 (<https://swagger.io/specification/>).
- [41] Michi Henning, *The Rise and Fall of CORBA*, „ACM Queue”, rocznik 4, nr 5, s. 28 – 34, czerwiec 2006 (<http://queue.acm.org/detail.cfm?id=1142044>; <https://dl.acm.org/citation.cfm?doid=1142031.1142044>).

- [42] Andrew D. Birrell i Bruce Jay Nelson, *Implementing Remote Procedure Calls*, „ACM Transactions on Computer Systems” (TOCS), rocznik 2, nr 1, s. 39 – 59, luty 1984 (<http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf>; <https://dl.acm.org/citation.cfm?doid=2080.357392>).
- [43] Jim Waldo, Geoff Wyant, Ann Wollrath i Sam Kendall, *A Note on Distributed Computing*, Sun Microsystems Laboratories, Inc., raport techniczny TR-94-29, listopad 1994 ([http://m.mirror.facebook.net/kde/devel/smli\\_tr-94-29.pdf](http://m.mirror.facebook.net/kde/devel/smli_tr-94-29.pdf)).
- [44] Steve Vinoski, *Convenience over Correctness*, „IEEE Internet Computing”, rocznik 12, nr 4, s. 89 – 92, lipiec 2008 ([http://steve.vinoski.net/pdf/IEEE-Convenience\\_Over\\_Correctness.pdf](http://steve.vinoski.net/pdf/IEEE-Convenience_Over_Correctness.pdf); <http://ieeexplore.ieee.org/document/4557985/>).
- [45] Marius Eriksen, *Your Server as a Function*, w: „7th Workshop on Programming Languages and Operating Systems” (PLOS), listopad 2013 (<https://monkey.org/~marius/funsrv.pdf>; <https://dl.acm.org/citation.cfm?doid=2525528.2525538>).
- [46] *grpc-common Documentation*, Google, Inc., [github.com](https://github.com/grpc/grpc-experiments), luty 2015 (<https://github.com/grpc/grpc-experiments>).
- [47] Aditya Narayan i Irina Singh, *Designing and Versioning Compatible Web Services*, [ibm.com](http://ibm.com), 28 marca 2007.
- [48] Troy Hunt, *Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways*, [troyhunt.com](http://troyhunt.com), 10 lutego 2014 (<https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/>).
- [49] *API Upgrades*, Stripe, Inc., kwiecień 2015 (<https://stripe.com/docs/upgrades>).
- [50] Jonas Bonér, *Upgrade in an Akka Cluster*, e-mail na liście dyskusyjnej [akka-user](http://akka-user.grokbases.com), [grokbase.com](http://grokbase.com), 28 sierpnia 2013 (<http://grokbase.com/t/gg/akka-user/138wd8j9e3/upgrade-in-an-akka-cluster>).
- [51] Philip A. Bernstein, Sergey Bykov, Alan Geller i in., *Orleans: Distributed Virtual Actors for Programmability and Scalability*, Microsoft Research Technical Report MSR-TR-2014-41, marzec 2014 (<https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F210931%2Forleans-msr-tr-2014-41.pdf>).
- [52] *Microsoft Project Orleans Documentation*, Microsoft Research, [dotnet.github.io](https://dotnet.github.io), 2015 (<http://dotnet.github.io/orleans/>).
- [53] David Mercer, Sean Hinde, Yinso Chen i Richard A O’Keefe, *beginner: Updating Data Structures*, wątek na liście dyskusyjnej [erlang-questions](http://erlang-questions.erlang.com), [erlang.com](http://erlang.com), 29 października 2007 (<http://erlang.org/pipermail/erlang-questions/2007-October/030318.html>).
- [54] Fred Hebert, *Postscript: Maps*, [learnyousoomeerlang.com](http://learnyousoomeerlang.com), 9 kwietnia 2014 (<http://learnyousoomeerlang.com/maps>).





W kierunku  
Spójności  
i Konsensusu  
(rozdział 9.)



W kierunku Baz o Strukturze  
Dziennika (rozdział 3.)

# Dane rozproszone

*Aby technologia odniosła sukces, rzeczywistość musi być ważniejsza od budowania wizerunku, ponieważ natury nie da się oszukać.*

— Richard Feynman, *Rogers Commission Report* (1986)

W części I tej książki omówiono aspekty systemów danych ważne w sytuacji, gdy dane są przechowywane na jednej maszynie. W części II przechodzimy na wyższy poziom i zadajemy pytanie: co się dzieje, gdy w składowaniu i pobieraniu danych bierze udział wiele maszyn?

Są różne powody, dla których można chcieć rozdzielić bazę między wiele maszyn:

## *Skalowalność*

Jeśli ilość danych albo obciążenie związane z odczytem lub zapisem rosną powyżej poziomu, jaki może zostać obsłużony przez jedną maszynę, można rozdzielić to obciążenie między wiele maszyn.

## *Odporność na błędy i wysoka dostępność*

Jeśli aplikacja musi kontynuować pracę nawet wtedy, gdy jedna maszyna (lub kilka maszyn, sieć albo całe centrum danych) przestanie działać, można zastosować więcej maszyn, aby zapewnić nadmiarowość. Gdy jedna maszyna zawiedzie, inna może przejąć jej funkcje.

## *Opóźnienie*

Jeśli użytkownicy znajdują się na całym świecie, możesz chcieć umieścić serwery w różnych lokalizacjach, tak aby móc obsłużyć każdego użytkownika za pomocą najbliższego mu geograficznie centrum danych. Dzięki temu użytkownicy nie muszą czekać na przesył pakietów sieciowych dookoła świata.

## Skalowanie pod kątem wyższego obciążenia

Jeśli potrzebujesz jedynie skalowania pod kątem wyższego obciążenia, najprostsze rozwiązanie to zakup mocniejszej maszyny (to podejście bywa czasem nazywane *skalowaniem pionowym*). Wiele procesorów, wiele układów pamięci RAM i wiele dysków można połączyć pod kontrolą jednego systemu operacyjnego, a szybkie połączenie umożliwia każdemu procesorowi dostęp do dowolnej

części pamięci lub dysku. W tego rodzaju *architekturze z pamięcią współużytkowaną* wszystkie komponenty można traktować jak jedną maszynę [1]<sup>1</sup>.

Problem z architekturą ze współużytkowaną pamięcią polega na tym, że koszty rosną wtedy szybciej niż liniowo. Maszyna, która ma dwa razy większą liczbę procesorów, dwa razy większą pamięć RAM i dwukrotnie wyższą pojemność dysku niż inna, zwykle kosztuje znacznie więcej niż dwukrotność ceny tej innej. Ponadto z powodu wąskich gardeł dwa razy mocniejsza maszyna nie zawsze sobie radzi z dwukrotnie wyższym obciążeniem.

Architektura ze współużytkowaną pamięcią może zapewniać ograniczoną odporność na błędy. Zaawansowane maszyny obejmują komponenty wymienne bez przerywania pracy systemu (można wymieniać dyski, moduły pamięci, a nawet procesory bez wyłączania urządzeń), jednak oczywiście są ograniczone do jednej lokalizacji geograficznej.

Inne podejście to *architektura ze współużytkowanymi dyskami*, gdzie kilka maszyn z niezależnymi procesorami i pamięcią RAM współużytkuje dane ze wspólnej macierzy dysków. Maszyny te są połączone szybką siecią<sup>2</sup>. Ta architektura jest używana dla niektórych rodzajów obciążenia roboczego w hurtowniach danych, jednak współzawodnictwo i koszty związane z blokadami ograniczają skalowalność w tym modelu [2].

## Architektury bez zasobów współdzielonych

Z kolei *architektury bez zasobów współdzielonych* [3] (ang. *shared-nothing*; ich inna nazwa to *skalowanie poziome*) zyskały znaczną popularność. W tym podejściu każda maszyna fizyczna lub wirtualna, na której działa oprogramowanie bazodanowe, jest nazywana *węzłem*. Każdy węzeł niezależnie korzysta ze swoich procesorów, pamięci RAM i dysków. Koordynacja pracy węzłów odbywa się na poziomie programowym z użyciem tradycyjnej sieci.

W systemie bez zasobów współdzielonych nie potrzeba specjalnego sprzętu. Możesz więc zastosować dowolne maszyny o najlepszym stosunku ceny do wydajności. Dane można rozdzielić między wiele obszarów geograficznych i ograniczyć w ten sposób opóźnienie odczuwalne przez użytkowników, a może nawet zapewnić odporność na utratę całego centrum danych. Dzięki instalowaniu maszyn wirtualnych w chmurze nie trzeba być firmą o wielkości Google'a. Nawet w małych firmach można obecnie stosować architektury rozproszone działające w wielu regionach.

Ta część książki dotyczy przede wszystkim architektur bez zasobów współdzielonych. Wynika to nie z tego, że w każdej sytuacji są najlepszym wyborem, ale z tego, że wymagają najwięcej ostrożności ze strony twórcy aplikacji. Jeśli dane są rozproszone między wiele węzłów, musisz mieć świadomość ograniczeń i kompromisów związanych z systemami rozproszonymi. Baza nie może w magiczny sposób ich przed Tobą ukryć.

---

<sup>1</sup> W dużych maszynach procesor wprawdzie może uzyskać dostęp do dowolnej części pamięci, jednak niektóre kości pamięci są bliżej jednych procesorów niż innych (jest to architektura NUMA; ang. *nonuniform memory access* [1]). Aby móc wydajnie korzystać z tej architektury, przetwarzanie trzeba rozdzielić w taki sposób, by każdy procesor używał głównie bliskiej mu pamięci. To oznacza, że podział i tak jest potrzebny, nawet gdy system działa na jednej maszynie.

<sup>2</sup> Są to sieci NAS (ang. *Network Attached Storage*) lub SAN (ang. *Storage Area Network*).

Choć rozproszona architektura bez zasobów współdzielonych ma wiele zalet, zwykle zwiększa złożoność aplikacji i czasem ogranicza możliwości modelu danych. W niektórych sytuacjach prosty program jednowątkowy może działać znacznie lepiej niż klastr z ponad setką rdzeni procesorów [4]. Systemy bez zasobów współdzielonych mogą jednak dawać bardzo duże możliwości. W kilku następnych rozdziałach szczegółowo opisane są problemy, jakie mogą wystąpić, gdy dane są rozproszone.

## Replikacja a podział

Istnieją dwa często stosowane sposoby rozmieszczania danych w wielu węzłach:

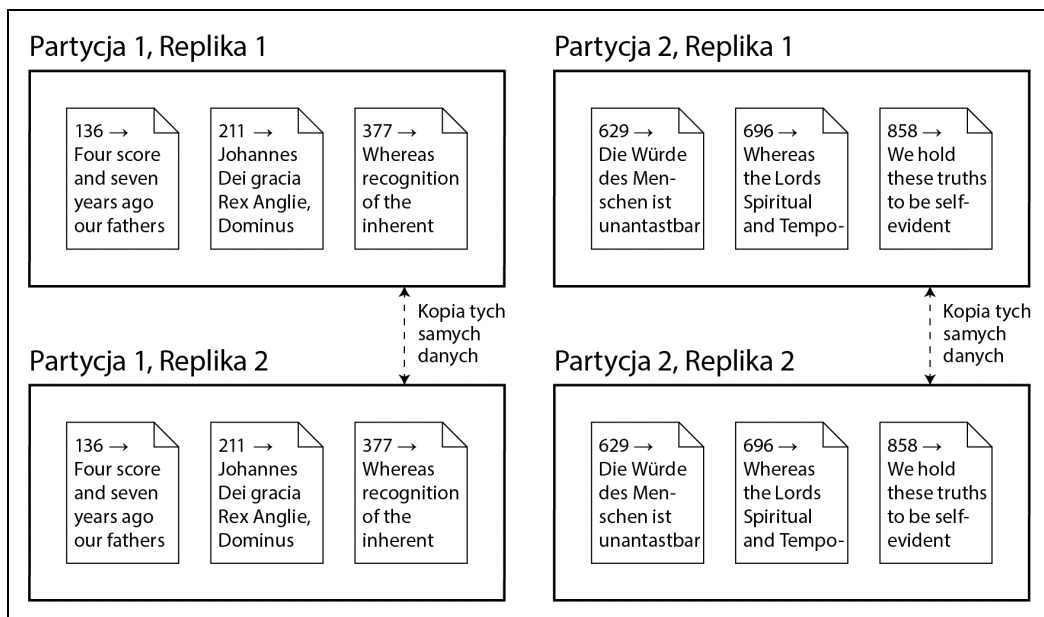
### Replikacja

Kopia tych samych danych jest umieszczana w różnych węzłach, czasem w różnych lokalizacjach. Replikacja zapewnia nadmiarowość. Jeśli niektóre węzły są niedostępne, dane i tak mogą być udostępniane przez pozostałe węzły. Replikacja może też pomóc w poprawie wydajności. Omówienie replikacji znajdziesz w rozdziale 5.

### Podział

Duża baza jest dzielona na mniejsze podzbiory nazywane *partycjami*. Poszczególne partycje można przypisywać do różnych węzłów. Inna nazwa tego podejścia to *sharding*. Podział na partycje opisano w rozdziale 6.

Są to odrębne mechanizmy, które jednak często stosuje się razem, co ilustruje rysunek II.1.



Rysunek II.1. Baza podzielona na dwie partycje z dwoma replikami każdej z nich

Po przedstawieniu tych zagadnień można omówić trudne kompromisy, jakie są konieczne w systemie rozproszonym. W rozdziale 7. opisane są *transakcje*, co pomoże Ci pojąć wiele różnych rzeczy, które mogą się nie powieść w systemie danych. Dowiesz się też, jak rozwiązać takie pro-

blemy. Ta część książki jest zakończona omówieniem podstawowych ograniczeń systemów rozproszonych (rozdziały 8. i 9.).

Dalej, w części III książki, opisano, jak zintegrować kilka (czasem rozproszonych) magazynów danych w większy system, aby spełnić potrzeby złożonej aplikacji. Najpierw jednak warto omówić dane rozproszone.

### **Literatura cytowana**

[1] Ulrich Drepper, *What Every Programmer Should Know About Memory*”, *akkadia.org*, 21 listopada 2007 (<https://www.akkadia.org/drepper/cpumemory.pdf>).

[2] Ben Stopford, *Shared Nothing vs. Shared Disk Architectures: An Independent View*, *benstopford.com*, 24 listopada 2009 (<http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/>).

[3] Michael Stonebraker, *The Case for Shared Nothing*, „IEEE Database Engineering Bulletin”, rocznik 9, nr 1, s. 4 – 9, marzec 1986 (<http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>).

[4] Frank McSherry, Michael Isard i Derek G. Murray, *Scalability! But at What COST?*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<http://www.frankmcsherry.org/assets/COST.pdf>).



W kierunku  
Spójności  
i Konsensusu  
(rozdział 9.)



W kierunku Baz o Strukturze  
Dziennika (rozdział 3.)



# Replikacja

*Główna różnica między tym, co może się zepsuć, a tym, co nie może się w żaden sposób zepsuć, polega na tym, że jeśli zepsuje się to, co nie może się w żaden sposób zepsuć, zwykle okazuje się, że nie można się do tego dostać ani tego naprawić.*

— Douglas Adams, *W zasadzie niegroźna* (1992)

*Replikacja* oznacza przechowywanie kopii tych samych danych na wielu maszynach połączonych siecią. We wprowadzeniu do części II opisano, że dane są replikowane z kilku powodów:

- W celu zachowania danych blisko użytkowników (geograficznie) i zmniejszenia w ten sposób opóźnień.
- Aby umożliwić systemowi kontynuowanie pracy mimo awarii niektórych jego części (co zwiększa dostępność).
- Aby zwiększyć liczbę maszyn, które mogą obsługiwać żądania odczytu (co zwiększa przepustowość odczytów).

W tym rozdziale przyjęto, że zbiór danych jest tak mały, iż każda maszyna może przechowywać kopię całego zbioru. W rozdziale 6., gdzie to założenie nie obowiązuje, opisano *podział* zbiorów danych, które są zbyt duże, aby zmieściły się na jednej maszynie. W dalszych rozdziałach omówiono różnego rodzaju błędy, które mogą występować w systemie danych z replikacją, oraz sposoby radzenia sobie z nimi.

Jeśli replikowane dane nie zmieniają się wraz z czasem, replikacja jest łatwa. Wystarczy raz skopiować dane do każdego węzła i gotowe. Cała trudność z replikacją polega na obsłudze *zmian* w replikowanych danych. Tego właśnie dotyczy ten rozdział. Omówione są tu trzy popularne algorytmy replikacji zmian w węzłach: *z jednym liderem*, *z wieloma liderami* i *bez lidera*. W prawie wszystkich rozproszonych bazach danych używane jest jedno z tych trzech podejść. Te techniki mają różne wady i zalety, które zostały tu szczegółowo przeanalizowane.

W kontekście replikacji trzeba uwzględnić wiele kompromisów związanych np. ze stosowaniem replikacji synchronicznej lub asynchronicznej i sposobem obsługi awarii replik. Często takie ustawienia wprowadza się za pomocą opcji konfiguracyjnych w bazie, a choć szczegóły zależą od bazy, ogólne zasady są takie same w wielu różnych rozwiązaniach. W tym rozdziale omówiono konsekwencje takich wyborów.

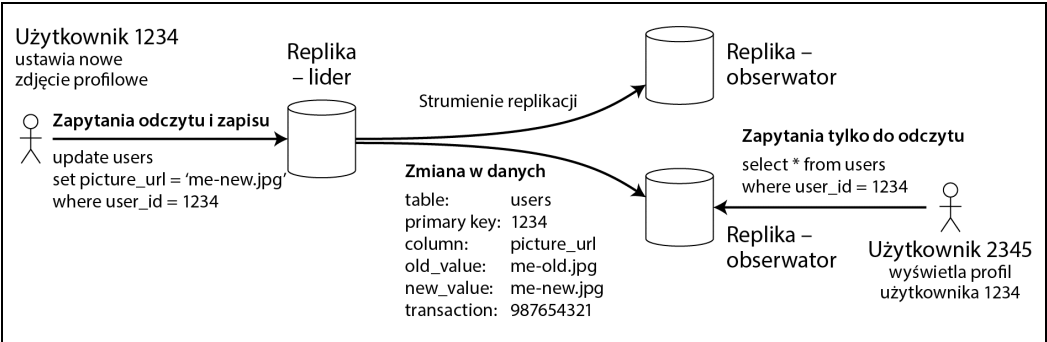
Temat replikacji baz jest znany od dawna. Jej zasady nie zmieniły się znacznie od czasów badań nad nimi w latach 70. [1], ponieważ podstawowe ograniczenia występujące w sieciach pozostają takie same. Jednak poza dziedziną badań wielu programistów przez długi czas przyjmowało, że baza obejmuje tylko jeden węzeł. Rozproszone bazy danych zaczęto powszechnie stosować później. Ponieważ dla wielu programistów aplikacji to nowy obszar, pojawiło się wiele nieporozumień wokół zagadnień takich jak *spójność ostateczna* (ang. *eventual consistency*). W punkcie „Problemy z opóźnieniem replikacji” spójność ostateczną opisano bardziej precyzyjnie. Omówiono tam także kwestie takie jak gwarancje spójności *odczytu własnych zapisów* (ang. *read-your-writes*) i *monotonności odczytów* (ang. *monotonic reads*).

## Liderzy i obserwatorzy

Każdy węzeł przechowujący kopię bazy nazywa się *repliką*. Gdy replik jest wiele, trzeba się zastanowić nad tym, jak sprawić, by w każdej replice znalazły się wszystkie dane.

Każdy zapis z bazy musi zostać przetworzony w każdej replice. W przeciwnym razie repliki nie będą zawierały tych samych danych. Najczęściej stosowane rozwiązanie to przedstawiona na rysunku 5.1 *replikacja oparta na liderze* (inne określenia to *replikacja aktywne-pasywne*). Działa ona tak:

1. Jedna z replik zostaje mianowana *liderem* (inaczej: repliką *nadrzędną* lub *główną*). Gdy klienci chcą zapisać dane w bazie, muszą przesłać zapytania do lidera, który najpierw zapisuje nowe dane w swoim lokalnym magazynie danych.
2. Pozostałe repliki są *obserwatorami* (inne nazwy to *replika do odczytu*, *replika podrzędna*, *replika pomocnicza* lub *rezerwa dynamiczna*). Gdy lider zapisuje nowe dane w lokalnym magazynie danych, przesyła do wszystkich powiązanych obserwatorów informacje o zmianach w ramach *dziennika replikacji* (inne określenie to *strumień zmian*). Każdy obserwator pobiera dziennik od lidera i odpowiednio aktualizuje lokalną kopię bazy, wprowadzając wszystkie zapisy w tej samej kolejności, w jakiej były przetwarzane u lidera.
3. Gdy klient chce wczytać dane z bazy, może skierować zapytanie albo do lidera, albo do jednego z obserwatorów. Jednak zapisy są akceptowane tylko przez lidera (z perspektywy klienta obserwatorzy są przeznaczeni tylko do odczytu).



Rysunek 5.1. Replikacja oparta na liderze (z replikami nadrzędną i podrzędnymi)

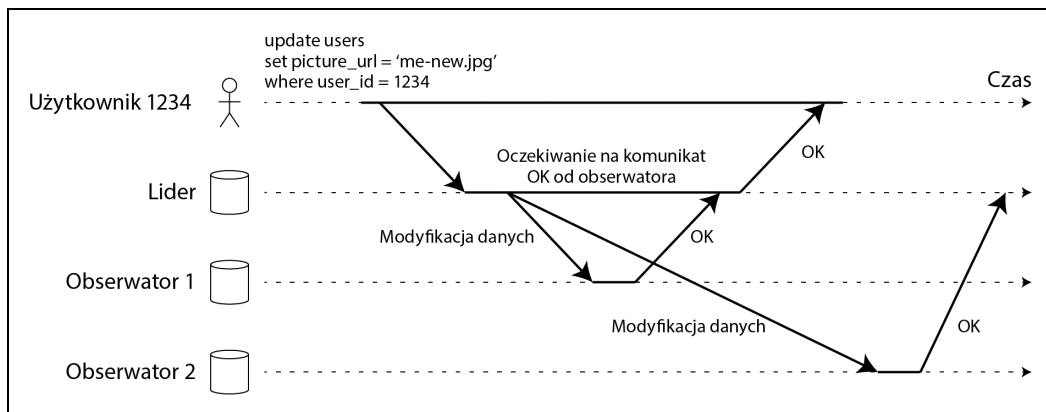
Replikacja w tym trybie jest wbudowana w wiele baz relacyjnych takich jak PostgreSQL (od wersji 9.0), MySQL, Oracle Data Ground [2] i AlwaysOn Availability Groups w SQL Server [3]. Jest używana także w niektórych bazach nierelacyjnych np. w MongoDB, RethinkDB i Espresso [4]. Poza tym replikacja oparta na liderze jest ograniczona nie tylko do baz. Korzystają z niej także rozproszone brokery komunikatów takie jak Kafka [5] i zapewniające wysoką dostępność kolejki RabbitMQ [6]. Podobnie działają niektóre sieciowe systemy plików i urządzenia blokowe (takie jak DRBD).

## Replikacja synchroniczna i asynchroniczna

Ważny aspekt systemów z replikacją stanowi to, czy odbywa się ona *synchronicznie* czy *asynchronicznie*. W bazach relacyjnych nieraz można to skonfigurować za pomocą opcji. W innych systemach często na stałe wbudowane jest jedno lub drugie rozwiązanie.

Pomyśl o tym, co się stanie w sytuacji z rysunku 5.1, gdy użytkownik witryny zaktualizuje swoje zdjęcie profilowe. W pewnym momencie klient przesyła do lidera żądanie aktualizacji. Krótko po tym lider otrzymuje to żądanie. W pewnej chwili lider przekazuje zmianę danych obserwatorom. Ostatecznie lider powiadamia klienta o powodzeniu aktualizacji.

Na rysunku 5.2 pokazano komunikację między różnymi komponentami systemu: klientem użytkownika, liderem i dwoma obserwatorami. Czas biegnie od lewej do prawej. Komunikaty z żądaniem lub odpowiedzią są wyróżnione grubą strzałką.



Rysunek 5.2. Replikacja oparta na liderze z jednym synchronicznym i jednym asynchronicznym obserwatorem

W przykładzie z rysunku 5.2 replikacja u obserwatora 1 odbywa się *synchronicznie*. Lider oczekuje na zatwierdzenie przez obserwatora 1 otrzymania zapisywanych danych, a dopiero potem informuje użytkownika o powodzeniu i udostępnia zapisane dane innym klientom. Replikacja u obserwatora 2 przebiega *asynchronicznie*. Lider przesyła komunikat, ale nie czeka na odpowiedź od tego obserwatora.

Na rysunku widać, że następuje znaczne opóźnienie do czasu przetworzenia komunikatu przez obserwatora 2. Zwykle replikacja przebiega dość szybko. Większość systemów bazodanowych wprowadza zmiany w obserwatorach w czasie krótszym niż sekunda. Nie ma jednak gwarancji, jak dużo

czasu to zajmie. W niektórych sytuacjach obserwatorzy mogą być opóźnieni względem lidera o kilka minut lub więcej. Dzieje się tak np. wtedy, gdy obserwator wznowia pracę po awarii, gdy system działa blisko granicy możliwości lub gdy występują problemy z siecią między węzłami.

Zaletą replikacji synchronicznej jest to, że obserwator ma gwarancję otrzymania aktualnej kopii danych spójnej z liderem. Jeśli lider nagle przestanie działać, można mieć pewność, że dane nadal będą dostępne u obserwatora. Wadą tego podejścia jest to, że jeśli synchroniczny obserwator nie zareaguje (z powodu jego awarii, błędu w sieci lub z innych przyczyn), operacji zapisu nie można przetworzyć. Lider musi zablokować wszystkie operacje zapisu i oczekiwać do momentu, gdy synchroniczna replika wznowi pracę.

Z tego powodu niepraktyczne jest, by wszyscy obserwatorzy byli synchroniczni, ponieważ przestój dowolnego węzła spowodowałby zatrzymanie całego systemu. W praktyce jeśli umożliwiasz synchroniczną replikację bazy, zwykle oznacza to, że *jeden* z obserwatorów działa synchronicznie, a pozostali są asynchroniczni. Gdy synchroniczny obserwator stanie się niedostępny lub powolny, jeden z asynchronicznych jest przekształcany w obserwatora synchronicznego. To gwarantuje, że aktualna kopia danych znajduje się w przynajmniej dwóch węzłach: liderze i jednym synchronicznym obserwatorze. Ta konfiguracja bywa czasem nazywana *semi-synchroniczną* [7].

Replikacja oparta na liderze jest czasem konfigurowana jako w pełni asynchroniczna. W takim scenariuszu jeśli lider przestanie działać i nie będzie można przywrócić jego stanu, wszystkie operacje zapisu, które nie zostały zreplikowane u obserwatorów, zostaną utracone. To oznacza, że nie ma gwarancji trwałości zapisu, nawet jeśli klient otrzymał potwierdzenie wykonania go. Jednak konfiguracja w pełni asynchroniczna ma tę zaletę, że lider może kontynuować przetwarzanie zapisów nawet wtedy, gdy wszyscy obserwatorzy są opóźnieni.

Oslabienie gwarancji trwałości może się wydawać kiepskim kompromisem, jednak replikacja asynchroniczna jest powszechnie stosowana — zwłaszcza w sytuacji, gdy występuje wielu obserwatorów lub gdy są oni geograficznie rozproszeni. Temat ten ponownie omówiono w punkcie „Problemy z opóźnieniem replikacji”.

## Badania nad replikacją

W systemach z asynchroniczną replikacją utrata danych po awarii systemu może być poważnym problemem. Dlatego badacze wciąż analizują metody replikacji, które nie prowadzą do utraty danych, ale i tak zapewniają wysoką wydajność i dostępność. Np. *replikacja łańcuchowa* [8, 9] to odmiana replikacji synchronicznej z powodzeniem zaimplementowana w różnych systemach, m.in. w Microsoft Azure Storage [10, 11].

Występuje ścisłe powiązanie między spójnością replikacji a *konsensem* (uzgadnianiem wartości przez grupę węzłów). Ten obszar teorii opisano szczegółowo w rozdziale 9. Niniejszy rozdział dotyczy prostszych form replikacji, które są najczęściej spotykane w bazach w praktyce.

## Tworzenie nowych obserwatorów

Od czasu do czasu trzeba skonfigurować nowych obserwatorów — np. w celu zwiększenia liczby replik lub zastąpienia węzłów po awarii. Jak się upewnić, że nowy obserwator obejmuje poprawną kopię danych lidera?

Proste skopiowanie pliku z danymi z jednego węzła do innego zwykle nie wystarcza. Klienci nieustannie zapisują dane w bazie, tak więc dane stale się zmieniają, dlatego standardowa kopia pliku będzie obejmowała różne części bazy z różnych momentów w czasie. Efekt może nie mieć żadnego sensu.

Możesz zapewnić spójność plików z dysku, blokując bazę (przez co staje się ona niedostępna do zapisu). To jednak jest sprzeczne z celem wysokiej dostępności. Na szczęście nowego obserwatora można zwykle dodać bez powodowania przestoju. W teorii ten proces wygląda tak:

1. Wykonanie spójnego snapshota bazy lidera z określonego czasu — w miarę możliwości bez blokowania całej bazy. Większość baz udostępnia tę funkcję, ponieważ jest potrzebna także do tworzenia kopii zapasowych. W niektórych sytuacjach konieczne są niezależne narzędzia takie jak *innobackupex* dla baz MySQL [12].
2. Skopiowanie snapshota do węzła z nowym obserwatorem.
3. Obserwator nawiązuje połączenie z liderem i żąda wszystkich zmian w danych wprowadzonych od momentu wykonania snapshota. To wymaga, aby snapshot był powiązany z określoną pozycją w dzienniku replikacji lidera. Ta pozycja bywa nazywana w różny sposób. Na przykład w bazie PostgreSQL jej nazwa to *numer sekwencyjny dziennika*, a w MySQL-u — *współrzędne dziennika binarnego*.
4. Gdy obserwator przetworzy zaległe zmiany w danych wprowadzone po wykonaniu snapshota, można powiedzieć, że *nadrobił zaległości*. Od tego momentu może zacząć przetwarzać zmiany w danych wraz z ich wprowadzaniem.

W praktyce kroki związane z tworzeniem obserwatora bardzo się różnią w zależności od baz. W niektórych systemach ten proces jest w pełni zautomatyzowany, natomiast w innych może to być zawiły, wieloetapowy proces, który musi zostać ręcznie wykonany przez administratora.

## Radzenie sobie z przestojami węzłów

Dowolny węzeł w systemie może zakończyć pracę — możliwe, że nieoczekiwanie (z powodu błędu), albo też z przyczyny planowanej konserwacji (np. przy ponownym rozruchu maszyny w celu zainstalowania poprawki bezpieczeństwa jądra). Możliwość ponownego rozruchu poszczególnych węzłów bez powodowania przestojów to duża zaleta w zakresie eksploatacji i konserwacji. Dlatego celem jest utrzymywanie sprawności całego systemu mimo awarii poszczególnych węzłów i sprawienie, by wpływ przestoju węzła był jak najmniejszy.

Jak uzyskać wysoką dostępność w modelu replikacji opartej na liderze?

## Awaria obserwatora — odzyskiwanie stanu z nadrabianiem zaległości

Na dysku lokalnym każdy obserwator przechowuje dziennik ze zmianami w danych, jakie otrzymał od lidera. Gdy obserwator się zepsuje i wznowi pracę lub gdy sieć między liderem a obserwatorem tymczasowo przestanie działać, obserwator może stosunkowo łatwo przywrócić stan. Dzięki dziennikowi obserwator zna ostatnią transakcję przetworzoną przed wystąpieniem awarii. Dlatego obserwator może nawiązać połączenie z liderem i zażądać wszystkich zmian w danych, jakie wystąpiły od momentu utraty połączenia. Wprowadzenie tych zmian oznacza nadrobienie zaległości. Następnie obserwator może tak jak wcześniej odbierać strumień zmian w danych.

## Awaria lidera — przełączanie awaryjne

Radzenie sobie z awarią lidera jest trudniejsze. Jednego z obserwatorów trzeba promować do funkcji lidera, klienty należy zrekonfigurować, aby przesyłały operacje zapisu do nowego lidera, a inni obserwatorzy muszą zacząć pobierać zmiany w danych od nowego lidera. Ten proces nazywa się *przełączaniem awaryjnym* (ang. *failover*).

Przełączanie awaryjne może być wykonywane ręcznie (administrator otrzymuje informację, że lider przestał działać, i podejmuje kroki niezbędne do utworzenia nowego lidera) lub automatycznie. Proces automatycznego przełączania awaryjnego zwykle obejmuje następujące kroki:

1. *Ustalenie, że lider przestał działać.* Kłopoty mogą wynikać z wielu powodów: awarii, braku zasilania, problemów z siecią itd. Nie istnieje uniwersalny sposób na wykrywanie tego, co poszło nie tak. Dlatego w większości systemów stosowane są limity czasu. Węzły często przekazują między sobą komunikaty, a gdy jeden węzeł nie odpowie przez określony czas (np. 30 s), przyjmuje się, że nie działa. Nie dotyczy to sytuacji, gdy lider jest celowo wyłączany na potrzeby planowanej konserwacji.
2. *Wybór nowego lidera.* Może się to odbywać w ramach wyborów (gdy lider jest ustalany przez większość pozostałych replik) lub w wyniku wskazania nowego lidera przez wcześniej określony *węzeł kontrolera*. Najlepszy kandydat do roli lidera to zwykle replika z najbardziej aktualnymi zmianami otrzymanymi od dawnego lidera (minimalizuje to utratę danych). Uzgadnianie nowego lidera przez wszystkie węzły to problem osiągnięcia konsensusu omówiony szczegółowo w rozdziale 9.
3. *Rekonfigurowanie systemu pod kątem używania nowego lidera.* Następnie klienty muszą zacząć przysyłać żądania zapisu do nowego lidera (ten temat opisano w punkcie „Trasowanie żądań”). Jeśli dawny lider wznowi pracę, może wciąż przyjmować, że jest liderem, nie wiedząc, że inne repliki pozbawiły go tej funkcji. System musi zapewnić, że dawny lider stanie się obserwatorem i wykryje nowego lidera.

Przełączanie awaryjne jest narażone na różne problemy:

- Jeśli stosowana jest replikacja asynchroniczna, nowy lider może nie otrzymać wszystkich operacji zapisu od dawnego lidera przed awarią tego ostatniego. Co zrobić z tymi operacjami, gdy dawny lider ponownie dołączy do klastra po wybraniu nowego? Nowy lider może w międzyczasie otrzymać operacje zapisu niezgodne z dawnymi. Najczęściej stosowane rozwiązanie polega na odrzuceniu niezreplikowanych zapisów od dawnego lidera, co jednak może naruszać oczekiwania klientów co do trwałości danych.

- Odrzucanie operacji zapisu jest niebezpieczne zwłaszcza w sytuacji, gdy inne systemy składowania danych (spoza bazy) muszą być koordynowane z zawartością bazy. Na przykład w trakcie awarii w serwisie GitHub [13] na lidera promowany został obserwator z nieaktualną bazą MySQL. W tej bazie do przypisywania kluczy głównych do nowych wierszy używany był automatycznie zwiększany licznik. Jednak ponieważ licznik w nowym liderze był opóźniony względem licznika z dawnego lidera, ponownie wykorzystał klucze główne przypisane wcześniej przez dawnego lidera. Te klucze główne były też używane w bazie Redis, dlatego ponowne wykorzystanie kluczy głównych spowodowało niespójności między bazami MySQL i Redis. Z tego powodu część prywatnych danych została ujawniona niewłaściwym użytkownikom.
- Czasem błędy powodują (zob. rozdział 8.), że dwa węzły jednocześnie sądzą, iż są liderem. Ta niebezpieczna sytuacja to *podzielony mózg*. Jeśli obaj liderzy akceptują operacje zapisu, a nie istnieje proces rozwiązywania konfliktów (zob. punkt „Replikacja z wieloma liderami”), dane zapewne zostaną utracone lub uszkodzone. W niektórych systemach używany jest bezpiecznik w formie mechanizm zamykającego jeden węzeł po wykryciu dwóch liderów<sup>1</sup>. Jeśli jednak ten mechanizm nie zostanie starannie zaprojektowany, efektem może być zamknięcie obu węzłów [14].
- Jaki jest odpowiedni limit czasu do uznania lidera za nieaktywnego? Dłuższy limit oznacza dłuższe przywracanie stanu po awarii. Jeśli jednak limit jest zbyt krótki, może skutkować niepotrzebnym przełączaniem awaryjnym. Na przykład tymczasowy skok obciążenia może sprawić, że czas odpowiedzi węzła wzrośnie powyżej limitu, a usterki w sieci mogą skutkować opóźnionymi pakietami. Jeżeli system już się zmagają z wysokim obciążeniem lub problemami w sieci, niepotrzebne przełączanie awaryjne zapewne pogorszy sytuację, zamiast ją naprawić.

Nie istnieją łatwe rozwiązania tych problemów. Dlatego niektóre zespoły eksploatacji wolą wykonywać przełączanie awaryjne ręcznie nawet wtedy, gdy oprogramowanie potrafi robić to automatycznie.

Wymienione kwestie (awarie węzłów, zawodna sieć i kompromisy związane ze spójnością, trwałością, dostępnością i opóźnieniami w replikach) są fundamentalnymi problemami systemów rozproszonych. W rozdziałach 8. i 9. tematy te są opisane bardziej szczegółowo.

## Implementowanie dzienników replikacji

W jaki sposób replikacja oparta na liderze działa na zapleczu? W praktyce stosuje się kilka różnych metod replikacji. Warto pokrótce opisać każdą z nich.

### Replikacja oparta na instrukcjach

W najprostszym rozwiązaniu lider rejestruje każde żądanie zapisu (*instrukcję*), które wykonuje, i przesyła dziennik instrukcji do obserwatorów. W bazach relacyjnych oznacza to, że do obserwatorów przekazywana jest każda instrukcja INSERT, UPDATE i DELETE, a każdy obserwator przetwarza i wykonuje te SQL-owe instrukcje w taki sposób, jakby otrzymał je od klienta.

---

<sup>1</sup> To podejście jest nazywane *odcięciem* (ang. *fencing*) lub dobitniej STONITH (ang. *Shoot The Other Node In The Head*, czyli „strzel drugiemu węzłowi w głowę”). Odcięcie jest omówione bardziej szczegółowo w punkcie „Lider i blokada”.

Choć to rozwiązanie może się wydawać sensowne, ta metoda replikacji może zawieść z kilku powodów:

- Każda instrukcja wywołująca funkcję niedeterministyczną (np. NOW() w celu uzyskania aktualnej daty i czasu lub RAND() w celu otrzymania liczby losowej) zapewne da inną wartość w każdej replice.
- Jeśli instrukcje używają kolumny z automatyczną inkrementacją lub zależą od istniejących danych z bazy (np. UPDATE ... WHERE <warunek>), muszą być wykonywane w każdej replice w tej samej kolejności (w przeciwnym razie mogą spowodować odmienne efekty). Może to być trudne do osiągnięcia, gdy równolegle wykonywanych jest wiele transakcji.
- Instrukcje dające efekty uboczne (np. wyzwalacze, procedury składowane, funkcje zdefiniowane przez użytkownika) mogą skutkować różnymi efektami ubocznymi w każdej replice, chyba że efekty te są w pełni deterministyczne.

Wymienione problemy można rozwiązać. Na przykład lider może w ramach rejestrowania instrukcji zastąpić dowolne niedeterministyczne wywołania funkcji stałą wartością, dzięki czemu wszyscy obserwatorzy otrzymają tę samą wartość. Jednak ponieważ istnieje tak wiele przypadków brzegowych, obecnie zwykle preferowane są inne metody replikacji.

Replikacja oparta na instrukcjach była stosowana w MySQL-u w wersjach starszych niż 5.1. Obecnie nadal czasem jest używana, ponieważ jest stosunkowo zwięzła. Jednak gdy instrukcje są niedeterministyczne, w MySQL-u obecnie domyślnie stosuje się opisaną dalej replikację opartą na wierszach. W bazie VoltDB używa się replikacji opartej na instrukcjach — to rozwiązanie jest bezpieczne dzięki wymogowi wykonywania deterministycznych transakcji [15].

## Przesyłanie dziennika WAL

W rozdziale 3. opisano, w jaki sposób systemy składowania danych reprezentują dane na dysku. Dowiedziałeś się tam, że zwykle każda operacja zapisu jest dodawana do dziennika:

- W systemach ze strukturą dziennika (zob. punkt „Pliki SSTable i drzewa LSM”) dziennik to główne miejsce składowania danych. W tle segmenty dziennika są kompresowane, a pamięć jest odzyskiwana.
- Gdy używane są b-drzewa (zob. punkt „B-drzewa”), które nadpisują poszczególne bloki na dysku, każda modyfikacja jest najpierw zapisywana w dzienniku WAL, dzięki czemu po awarii można przywrócić indeks w spójnym stanie.

W obu sytuacjach dziennik to umożliwiająca tylko dodawanie danych sekwencja bajtów zawierająca wszystkie operacje zapisu w bazie. Ten sam dziennik można wykorzystać do utworzenia repliki w innym węźle. Oprócz zapisu dziennika na dysku lider przesyła go też siecią do obserwatorów. Gdy obserwator przetwarza dziennik, tworzy kopię tych samych struktur danych, które znajdują się u lidera.

Ta metoda replikacji jest stosowana m.in. w bazach PostgreSQL i Oracle [16]. Główną wadę tego podejścia stanowi to, że dziennik opisuje dane na bardzo niskim poziomie. Dziennik WAL zawiera szczegóły określające, które bajty zostały zmodyfikowane w poszczególnych blokach dysku. Z tego powodu replikacja jest ściśle powiązana z systemem składowania danych. Jeśli w bazie format składowania zostanie zmieniony z jednej wersji na inną, zwykle nie da się używać różnych wersji oprogramowania bazodanowego u lidera i u obserwatorów.



Może się to wydawać drobnym szczegółem implementacji, może jednak mieć istotny wpływ na eksploatację. Jeśli protokół replikacji umożliwia obserwatorowi zastosowanie nowszej wersji oprogramowania niż u lidera, można zaktualizować oprogramowanie bazodanowe bez przestojów. W tym celu najpierw należy zaktualizować obserwatorów, a następnie przeprowadzić przełączanie awaryjne, aby jeden ze zaktualizowanych węzłów stał się nowym liderem. Jeśli protokół replikacji nie pozwala na takie niedopasowanie wersji (co zdarza się często, gdy używa się przesyłania dzienników WAL), taka aktualizacja wymaga przestoju.

### Replikacja z użyciem dziennika logicznego (oparta na wierszach)

Inna możliwość to stosowanie różnych formatów dzienników dla replikacji i dla systemu składowania danych. Pozwala to oddzielić dziennik replikacji od wewnętrznych mechanizmów systemu składowania danych. Tego rodzaju dziennik replikacji jest nazywany *dziennikiem logicznym* (w celu odróżnienia go od *fizycznej* reprezentacji danych z systemu składowania).

Dziennik logiczny dla baz relacyjnych to zwykle sekwencja rekordów reprezentujących operacje zapisu w tabeli bazy na poziomie wierszy:

- Dla wstawianych wierszy dziennik obejmuje nowe wartości wszystkich kolumn.
- Dla usuwanych wierszy dziennik zawiera informacje wystarczające do unikatowego zidentyfikowania usuwanego wiersza. Zwykle te informacje to klucz główny, jeśli jednak tabela go nie ma, trzeba zarejestrować dawne wartości wszystkich kolumn.
- Dla aktualizowanych wierszy dziennik zawiera informacje wystarczające do unikatowego zidentyfikowania zaktualizowanego wiersza oraz nowe wartości wszystkich kolumn (a przynajmniej nowe wartości zmodyfikowanych kolumn).

Transakcja modyfikująca kilka wierszy generuje kilka rekordów dziennika, po których następuje rekord z informacją o zatwierdzeniu transakcji. To podejście jest używane w dzienniku binarnym z bazy MySQL (gdy baza została skonfigurowana, by używać replikacji opartej na wierszach) [17].

Ponieważ dziennik logiczny jest oddzielony od wewnętrznych mechanizmów systemu składowania danych, pozwala łatwiej zachować zgodność wstecz i umożliwia liderowi oraz obserwatorowi korzystanie z różnych wersji oprogramowania bazodanowego, a nawet różnych systemów składowania danych.

Format dziennika logicznego jest też łatwiejszy do przetwarzania w aplikacjach zewnętrznych. Ta cecha może być przydatna, jeśli zawartość bazy chcesz przesyłać do systemu zewnętrznego (np. do hurtowni danych na potrzeby analiz offline) lub wykorzystać do tworzenia niestandardowych indeksów i pamięci podręcznych [18]. Ta technika to *rejestrwanie zmian w danych*. Wrócimy do niej w rozdziale 11.

### Replikacja oparta na wyzwalaczach

Techniki replikacji opisane do tej pory są implementowane w systemie bazodanowym bez udziału kodu aplikacji. W wielu sytuacjach to pożądane rozwiązanie, jednak w niektórych okolicznościach potrzebna jest większa elastyczność. Na przykład jeśli chcesz replikować tylko podzbiór danych lub dane z bazy jednego rodzaju do innej albo potrzebujesz logiki rozwiązywania konfliktów (zob. punkt „Radzenie sobie z konfliktami przy zapisie”), konieczne może być przeniesienie replikacji do warstwy aplikacji.

Niektóre narzędzia, np. Oracle GoldenGate [19], mogą udostępniać aplikacji zmiany w danych na podstawie odczytu dziennika bazy danych. Inną możliwość to wykorzystanie mechanizmów dostępnych w wielu bazach relacyjnych: *wyzwalaczy* i *procedur składowanych*.

Wyzwalacz umożliwia zarejestrowanie niestandardowego kodu aplikacji, który jest automatycznie wykonywany po zmianie danych (po transakcji zapisu) w systemie bazodanowym. Wyzwalacz może zapisywać tę zmianę w odrębnej tabeli, z której dane wczytuje zewnętrzny proces. Ten zewnętrzny proces może uruchamiać niezbędną logikę aplikacji i replikować zmiany danych w innym systemie. Tak działają np. Databus dla baz Oracle [20] i Bucardo dla baz Postgres [21].

Replikacja oparta na wyzwalaczach powoduje zwykle większe koszty niż inne techniki replikacji i jest bardziej narażona na błędy i ograniczenia niż wbudowana replikacja z bazy danych. Mimo to dzięki elastyczności może okazać się użyteczna.

## Problemy z opóźnieniem replikacji

Odporność na awarie węzła to tylko jeden z powodów stosowania replikacji. We wprowadzeniu do części II wspomniano, że innymi przyczynami są skalowalność (przetwarzanie większej liczby żądań niż jest to możliwe w jednej maszynie) i opóźnienie (rozmieszczanie replik geograficznie bliżej użytkowników).

Replikacja oparta na liderze wymaga, by wszystkie operacje zapisu przechodziły przez jeden węzeł. Jednak zapytania z samym odczytem można kierować do dowolnej repliki. Gdy obciążenie robocze obejmuje głównie odczyty i tylko niewielki odsetek zapisów (w internecie to częsty wzorzec), istnieje atrakcyjna możliwość: utworzenie wielu obserwatorów i rozdzielanie żądań odczytu między nich. Pozwala to zmniejszyć obciążenie lidera i umożliwia obsługę żądań odczytu przez repliki blisko użytkowników.

W architekturze *ze skalowaniem odczytów* można zwiększyć przepustowość obsługi żądań z samym odczytem w prosty sposób — dodając obserwatorów. Jednak to podejście jest wykonalne tylko w replikacji asynchronicznej. Jeśli spróbujesz synchronicznie replikować dane we wszystkich obserwatorach, awaria jednego węzła lub sieci sprawi, że zapis stanie się niedostępny w całym systemie. A im więcej węzłów, tym większe prawdopodobieństwo, że któryś z nich przestanie działać. Dlatego w pełni synchroniczna konfiguracja jest bardzo niestabilna.

Niestety, jeśli aplikacja wczytuje dane z *asynchronicznego* obserwatora, może otrzymać przestarzałe informacje, gdy dany obserwator jest opóźniony. To prowadzi do niespójności w bazie. Jeśli w tym samym momencie uruchomisz to samo zapytanie u lidera i u obserwatora, możesz otrzymać różne wyniki, ponieważ u obserwatora nie wszystkie zapisy zostały uwzględnione. Ta niespójność jest tymczasowa. Jeżeli zaprzestaniesz zapisu w bazie i odczekasz pewien czas, obserwatorzy ostatecznie nadrobą zaległości względem lidera i staną się z nim spójni. Dlatego ten efekt nazywa się *spójnością ostateczną* [22, 23]<sup>2</sup>.

---

<sup>2</sup> Pojęcie *spójność ostateczna* zostało wymyślone przez Douglasa Terry'ego i współpracowników [24], a spopularyzowane przez Wernera Vogelsa [22]. Stało się też „hasłem bojowym” w wielu projektach NoSQL. Jednak nie tylko bazy NoSQL cechują się spójnością ostateczną. Obserwatorzy w bazach relacyjnych z replikacją asynchroniczną też mają tę właściwość.

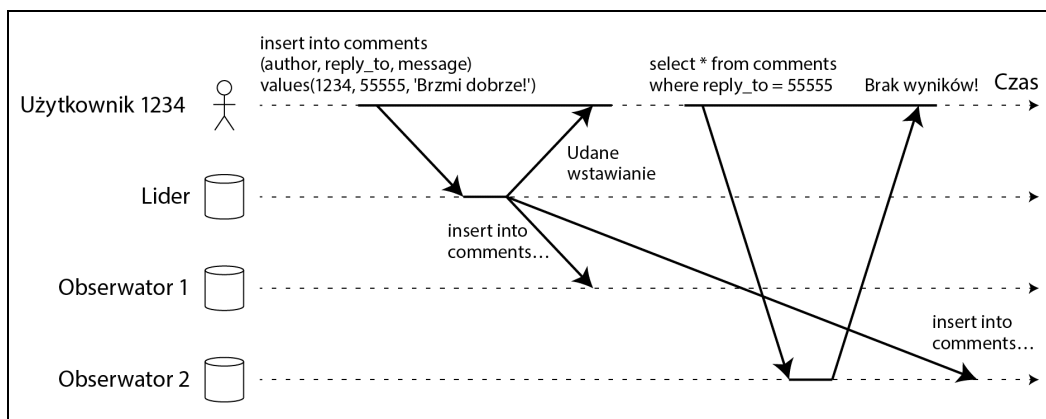
Słowo „ostateczna” jest celowo niejednoznaczne. Nie istnieje limit określający, jak duże może być opóźnienie repliki. W trakcie normalnej eksploatacji opóźnienie między zapisem u lidera a odzwierciedleniem tej operacji u obserwatora (*opóźnienie replikacji*) może wynosić tylko ułamek sekundy i w praktyce być nieodczuwalne. Jeśli jednak system działa na granicy możliwości lub gdy występuje problem w sieci, opóźnienie może łatwo wzrosnąć do wielu sekund, a nawet minut.

Gdy opóźnienie jest tak duże, powodowane przez nie niespójności nie stanowią tylko kwestii teoretycznej, ale rzeczywisty problem dla aplikacji. W tym punkcie omówione zostały trzy przykładowe problemy, które mogą wystąpić z powodu opóźnienia replikacji. Przedstawione są też różne sposoby ich rozwiązywania.

## Odczyt własnych zapisów

Wiele aplikacji umożliwia użytkownikowi przesłanie danych i późniejsze ich wyświetlenie. Tymi danymi może być rekord w bazie klientów, komentarz w wątku dyskusyjnym lub coś podobnego. Gdy dane są przesyłane, muszą trafić do lidera. Jednak gdy użytkownik wyświetla dane, mogą one zostać wczytane z obserwatora. To dobre rozwiązanie zwłaszcza wtedy, gdy dane są często wyświetlane, ale rzadko zapisywane.

Z replikacją asynchroniczną wiąże się problem przedstawiony na rysunku 5.3. Jeśli użytkownik wyświetla dane krótko po ich zapisie, nowe dane mogły nie dotrzeć jeszcze do repliki. Dla użytkownika wygląda to tak, jakby przesłane dane zostały utracone. Zrozumiałe jest więc niezadowolenie użytkownika.



Rysunek 5.3. Użytkownik zapisuje dane, po czym dokonuje odczytu z przestarzałej repliki. Aby zapobiec takiej anomalii, potrzebna jest spójność odczytu po zapisie

W takiej sytuacji potrzebna jest *spójność odczytu po zapisie* nazywana też *spójnością odczytu własnych zapisów* [24]. Zgodnie z tą gwarancją jeśli użytkownik odświeży stronę, zawsze zobaczy wszystkie aktualizacje, które sam przesłał. Ta gwarancja nie dotyczy innych użytkowników. Aktualizacje innych osób mogą się stać widoczne dopiero po pewnym czasie. Ten model daje jednak użytkownikom pewność, że ich własne dane wejściowe zostały poprawnie zachowane.

Jak można zapewnić spójność odczytu po zapisie w systemie z replikacją opartą na liderze? Dostępne są różne techniki. Oto kilka z nich:

- Odczyt danych, które mógł zmodyfikować dany użytkownik, jest obsługiwany za pomocą lidera. Pozostałe odczyty są obsługiwane przez obserwatora. To wymaga sposobu wykrywania możliwych modyfikacji bez zgłaszania zapytań. Na przykład informacje w profilu użytkownika w sieci społecznościowej zwykle może modyfikować tylko właściciel tego profilu i nikt inny. Dlatego prosta reguła brzmi: zawsze wczytuj własny profil użytkownika z lidera, a profile innych użytkowników z obserwatora.
- Jeśli użytkownik może edytować większość danych w aplikacji, opisane podejście jest nieefektywne, ponieważ większość informacji trzeba wtedy wczytywać z lidera (co eliminuje korzyści ze skalowania odczytu). W takiej sytuacji można się posłużyć innymi kryteriami do decydowania, czy dane należy wczytać z lidera. Możesz np. śledzić czas ostatniej aktualizacji i przez minutę od jej przeprowadzenia wszystkie dane wczytywać z lidera. Możesz też monitorować opóźnienie replikacji u obserwatorów i nie kierować zapytań do obserwatorów opóźnionych więcej niż o minutę względem lidera.
- Klient może zapamiętywać znacznik czasu ostatniego zapisu. Wtedy system może gwarantować, że replika obsługująca odczyt tego użytkownika obejmuje aktualizacje przynajmniej do tego czasu. Jeśli replika nie jest wystarczająco aktualna, odczyt można obsłużyć za pomocą innej repliki lub zapytanie może oczekiwać do czasu nadrobienia zaległości przez daną replikę. Można tu używać *logicznego znacznika czasu* (czegoś, co określa kolejność zapisów; może to być numer porządkowy w dzienniku) lub rzeczywistego czasu z zegara systemowego (wtedy niezbędna jest synchronizacja zegarów; zob. punkt „Zawodne zegary”).
- Jeśli repliki są rozproszone między wiele centrów danych (w celu zapewnienia dostępności lub geograficznej bliskości względem użytkowników), złożoność dodatkowo rośnie. Wszystkie żądania, które powinien obsłużyć lider, muszą być kierowane do centrum danych zawierającego lidera.

Inna komplikacja powstaje, gdy ten sam użytkownik korzysta z usługi na różnych urządzeniach — np. w przeglądarce w komputerze stacjonarnym i w aplikacji mobilnej. Wtedy można zapewnić spójność odczytu po zapisie *dla wielu urządzeń*. Jeśli użytkownik wprowadzi informacje w jednym urządzeniu, a następnie wyświetli je w innym, powinien zobaczyć wpisane dane.

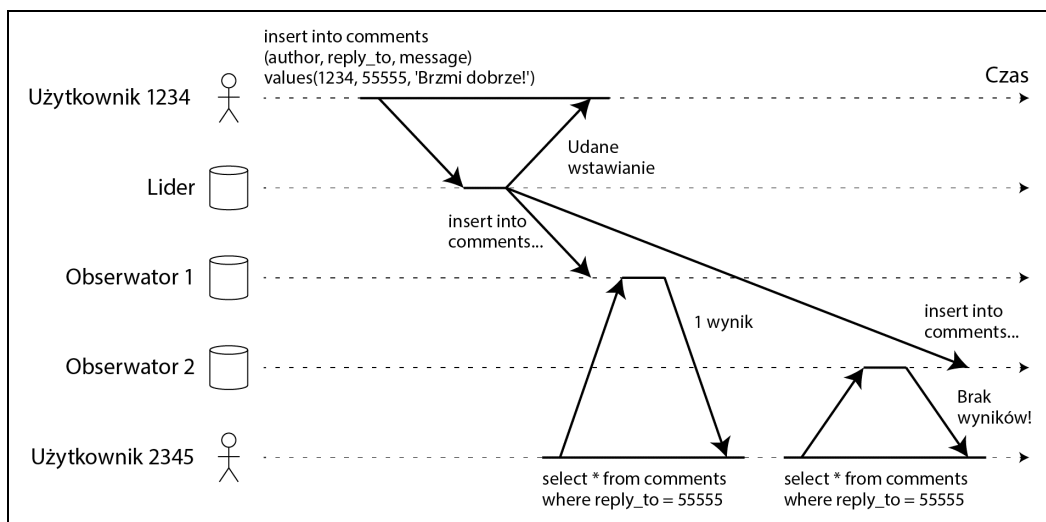
W tym scenariuszu trzeba uwzględnić dodatkowe kwestie:

- Techniki wymagające zapamiętywania znacznika czasu ostatniej aktualizacji dokonanej przez użytkownika stają się trudniejsze, ponieważ kod z jednego urządzenia nie wie, jakie aktualizacje zostały wprowadzone w innym. Te metadane trzeba scentralizować.
- Jeśli repliki są rozproszone między różne centra danych, nie ma gwarancji, że połączenia z różnych urządzeń zostaną skierowane do tego samego centrum danych. Na przykład jeśli komputer stacjonarny użytkownika korzysta z domowego połączenia szerokopasmowego, a urządzenie przenośne posługuje się siecią komórkową, trasy w sieci mogą być zupełnie inne. Jeśli technika wymaga odczytu danych z lidera, możliwe, że żądania z wszystkich urządzeń użytkownika trzeba będzie kierować do tego samego centrum danych.

## Odczyty monotoniczne

Druga przykładowa anomalia, która może wystąpić w trakcie odczytu danych z asynchronicznych obserwatorów, polega na tym, że użytkownik może zobaczyć dane *cofające się w czasie*.

Jest to możliwe, jeśli użytkownik wykonuje kilka odczytów z różnych replik. Na przykład na rysunku 5.4 pokazano, że użytkownik 2345 wykonuje to samo zapytanie dwukrotnie. Najpierw zgłasza je do obserwatora o niewielkim opóźnieniu, a następnie do obserwatora o większym opóźnieniu. Ten scenariusz jest całkiem prawdopodobny, jeśli użytkownik odświeża stronę, a każde żądanie jest kierowane do losowo wybranego serwera. Pierwsze zapytanie zwraca komentarz dodany niedawno przez użytkownika 1234. Jednak drugie zapytanie nie zwraca żadnych danych, ponieważ obserwator działający z opóźnieniem nie pobrał jeszcze zapisanych danych. W efekcie w drugim zapytaniu uwzględniany jest system z wcześniejszego momentu w czasie niż w pierwszym zapytaniu. Nie byłoby to tak złe, gdyby pierwsze zapytanie nie zwróciło żadnych danych, ponieważ użytkownik 2345 zapewne nie wiedziałby wtedy, że użytkownik 1234 przed chwilą dodał komentarz. Jednak bardzo myląca jest sytuacja, gdy użytkownik 2345 najpierw widzi, że komentarz użytkownika 1234 się pojawia, a później komentarz ten przestaje być widoczny.



Rysunek 5.4. Użytkownik najpierw wczytuje dane z aktualnej repliki, a następnie z nieaktualnej.

Wydaje się wtedy, że czas się cofa. Aby zapobiec tej anomalii, potrzebne są monotoniczne odczyty

*Monotoniczne odczyty* [23] gwarantują, że tego rodzaju anomalie nie będą się pojawiać. To słabsza gwarancja niż spójność silna, jednak silniejsza niż spójność ostateczna. Gdy wczytujesz dane, możesz zobaczyć starszą wartość. Monotoniczne odczyty oznaczają tylko tyle, że jeśli jeden użytkownik wykonuje kilka odczytów po kolei, nie widzi czasu bieżącego wstecz (czyli nie wczytuje starszych danych, jeśli wcześniej wczytał nowsze).

Jednym ze sposobów zapewnienia monotonicznych odczytów jest upewnienie się, że każdy użytkownik zawsze wczytuje dane z tej samej repliki (różni użytkownicy mogą wczytywać je z różnych replik). Replika może być wybierana np. na podstawie skrótu identyfikatora użytkownika zamiast losowo. Jeśli jednak ta replika jest niedostępna, zapytania użytkownika trzeba przekierować do innej.

## Odczyty ze spójnym przedrostkiem

Trzeci przykład anomalii związanych z opóźnieniem replikacji polega na naruszeniu przyczynowości. Wyobraź sobie następujący krótki dialog między panem Poonsem a panią Cake:

## Pan Poons

Jak daleko w przyszłość potrafi pani zajrzeć, pani Cake?

### *Pani Cake*

Zwykle na około dziesięć sekund, panie Poons.

Miedzy tymi dwoma zdaniemmi zachodzi zależność przyczynowa. Pani Cake usłyszała pytanie pana Poonsa i odpowiedziała na nie.

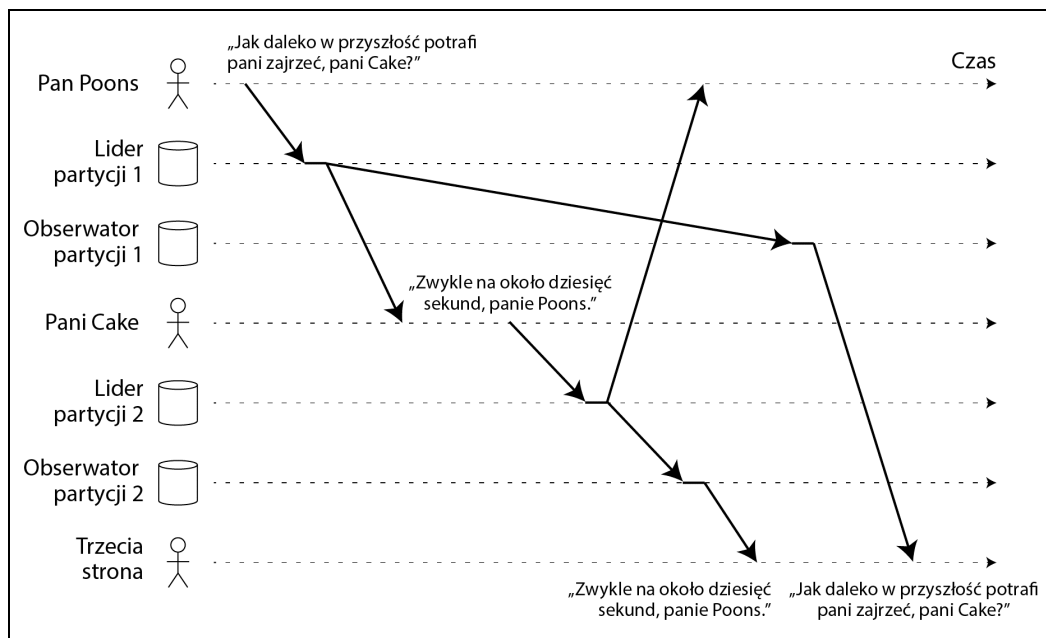
Teraz wyobraź sobie, że trzecia osoba słucha tej konwersacji za pośrednictwem obserwatorów. To, co mówi pani Cake, jest przekazywane przez obserwatorów z niewielkim opóźnieniem, jednak opóźnienie replikacji dla słów pana Poonsa jest większe (zob. rysunek 5.5). Trzecia strona może wtedy usłyszeć następujący dialog:

## Pani Cake

Zwykle na około dziesięć sekund, panie Poons.

## Pan Poons

Jak daleko w przyszłość potrafi pani zajrzeć, pani Cake?



*Rysunek 5.5. Jeśli niektóre partycje są replikowane wolniej od innych, trzecia strona może zobaczyć odpowiedź wcześniej niż pytanie*

Dla trzeciej strony wygląda to tak, jakby pani Cake odpowiedziała na pytanie, zanim pan Poons w ogóle je zadał. Takie moce psychiczne robią duże wrażenie, ale bardzo dezorientują [25].

Zapobieganie anomalii tego rodzaju wymaga gwarancji innego typu: *odczytów ze spójnym przedrostkiem* [23]. Ta gwarancja oznacza, że jeśli sekwencja zapisów ma określoną kolejność, każdy wczytujący te zapisy zobaczy je w tym samym porządku.

Opisana anomalia stanowi problem zwłaszcza w dzielonych bazach, których omówienie zawiera rozdział 6. Jeśli baza zawsze zapisuje dane w tej samej kolejności, odczyty zawsze mają spójny przedrostek, dlatego ta anomalia nie może wystąpić. Jednak w wielu bazach rozproszonych różne partycje działają niezależnie od siebie. Nie istnieje więc globalne uporządkowanie zapisów. Gdy użytkownik wczytuje dane z bazy, może zobaczyć niektóre jej części w dawnym stanie, a niektóre w nowym.

Jedno z rozwiązań to upewnienie się, że wszystkie zapisy przyczynowo powiązane ze sobą są zapisywane w tej samej partycji. Jednak w niektórych aplikacjach nie da się wydajnie uzyskać tego efektu. Istnieją też algorytmy, które bezpośrednio śledzą zależności przyczynowe. Do tego tematu wracamy w punkcie „Zależność »wystąpiło wcześniej« a współbieżność”.

## Rozwiązania problemu opóźnienia replikacji

W trakcie pracy z systemem ostatecznie spójnym warto się zastanowić nad tym, jak aplikacja działa, gdy opóźnienie replikacji wzrasta do kilku minut, a nawet godzin. Jeśli taka sytuacja nie stanowi problemu, to świetnie. Jeżeli jednak skutkuje złymi doświadczeniami dla użytkowników, ważne jest, aby zaprojektować system zapewniający mocniejsze gwarancje (np. odczytu po zapisie). Udawanie, że replikacja jest synchroniczna, podczas gdy w rzeczywistości jest asynchroniczna, to przepis na problemy.

Wcześniej opisano, że istnieją sposoby zapewniania przez aplikację mocniejszych gwarancji, niż oferuje je używana baza. Można np. wykonywać określone rodzaje odczytów za pomocą lidera. Jednak obsługa takich kwestii w kodzie aplikacji jest złożona i łatwo przy tym o błąd.

Byłoby lepiej, gdyby programiści aplikacji nie musieli się martwić o subtelne problemy z replikacją i mogli zaufać, że bazy „będą robiły to, co do nich należy”. To dlatego istnieją *transakcje*. Są one metodą, dzięki której baza może zapewnić silniejsze gwarancje, co pozwala uprościć aplikacje.

Transakcje w ramach jednego węzła istnieją od dawna. Jednak w ramach przechodzenia w kierunku baz rozproszonych (z replikacją i partycjami) w wielu systemach zrezygnowano z transakcji, twierdząc, że zaniechęcają negatywnie wpływają na wydajność i dostępność, oraz uznając, że w skalowalnym systemie nieuniknione jest ograniczenie się do spójności ostatecznej. Jest w tym trochę prawdy, ale to nadmierne uproszczenie. Do tematu transakcji wrócimy w rozdziałach 7. i 9., a w części III opisane są inne mechanizmy.

## Replikacja z wieloma liderami

Do tej pory w tym rozdziale omawiano wyłącznie architektury replikacji z użyciem jednego lidera. Choć to popularne podejście, istnieją też inne ciekawe rozwiązania.

Replikacja oparta na liderze ma jedną ważną wadę: występuje tylko jeden lider, a wszystkie zapisy muszą przez niego przechodzić<sup>3</sup>. Jeśli z jakiegś przyczyny nie możesz nawiązać połączenia z liderem (np. z powodu zakłóceń w sieci między Tobą a liderem), zapis danych w bazie nie jest możliwy.

Naturalnym rozwinięciem modelu replikacji opartej na liderze jest umożliwienie obsługi zapisów więcej niż jednemu węzłowi. Replikacja odbywa się wtedy w ten sam sposób: każdy węzeł, który przetwarza zapis, musi przekazywać zmiany danych do wszystkich pozostałych węzłów. To rozwiązanie jest nazywane konfiguracją z *wieloma liderami* (inne określenia to *nadrzędne-nadrzędne* lub *replikacja aktywne-aktywne*). W tym podejściu każdy lider działa jednocześnie jak obserwator innych liderów.

## Przypadki użycia replikacji z wieloma liderami

W ramach jednego centrum danych stosowanie konfiguracji z wieloma liderami rzadko ma sens, ponieważ korzyści rzadko przeważają nad wzrostem złożoności. Jednak w niektórych sytuacjach takie rozwiązanie jest przydatne.

### Eksploatacja wielu centrów danych

Wyobraź sobie, że używasz bazy z replikami w kilku różnych centrach danych (np. po to, by uzyskać odporność na awarię całego centrum danych lub być bliżej użytkowników). W normalnej replikacji opartej na liderze lider musi się znajdować w *jednym* centrum danych i wszystkie zapisy muszą przechodzić przez to centrum.

W konfiguracji z wieloma liderami można utworzyć lidera w *każdym* centrum danych. Na rysunku 5.6 pokazano, jak może wyglądać taka architektura. W ramach każdego centrum danych stosowana jest standardowa replikacja typu lider – obserwator. Między centrami danych lider z każdego centrum replikuje zmiany, przesyłając je do liderów z innych centrów.

Porównajmy teraz, jak konfiguracje z jednym liderem i z wieloma liderami sprawdzają się w środowisku z wieloma centrami danych.

#### Wydajność

W konfiguracji z jednym liderem każdy zapis trzeba przesłać internetem do centrum danych, gdzie działa ten lider. Może to znacznie zwiększyć opóźnienie zapisów i zniweczyć sens używania wielu centrów danych. W konfiguracji z wieloma liderami każdy zapis można przetwarzać w lokalnym centrum i asynchronicznie replikować w innych centrach. Dlatego opóźnienie sieciowe między centrami danych jest ukryte przed użytkownikami, co oznacza, że odczuwalna wydajność może być wyższa.

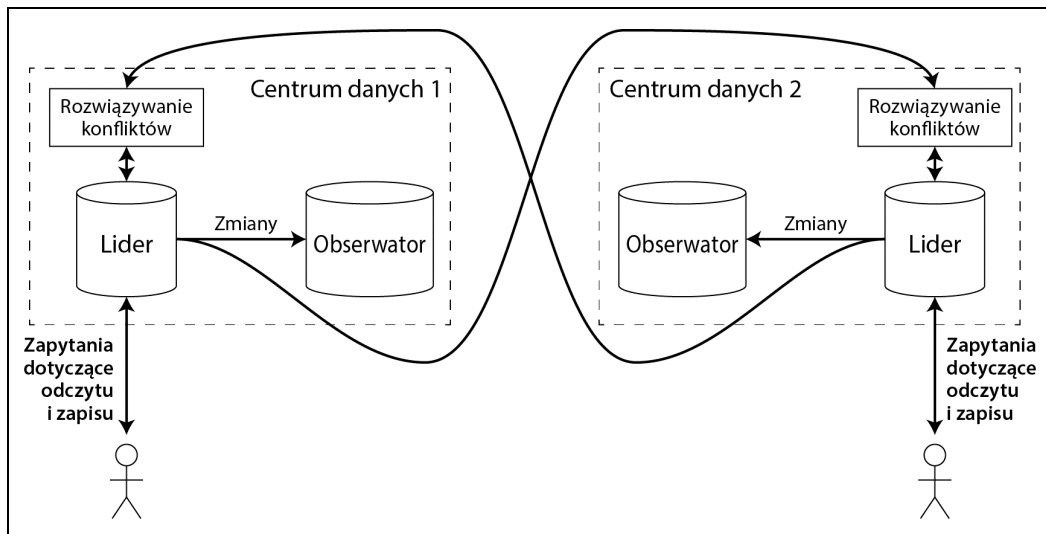
#### Odporność na przestoje centrów danych

Jeśli w konfiguracji z jednym liderem centrum danych obejmujące tego lidera zawiedzie, przełączanie awaryjne może spowodować promowanie na lidera obserwatora z innego centrum. W konfiguracji z wieloma liderami każde centrum danych może kontynuować pracę niezależnie od pozostałych, a replikacja pozwala zaktualizować stan, gdy uszkodzone centrum danych wznowi pracę.

---

<sup>3</sup> Jeśli baza jest podzielona na partycje (zob. rozdział 6.), każda partycja ma jednego lidera. Liderzy różnych partycji mogą się znajdować w innych węzłach, ale każda partycja musi mieć jeden węzeł lidera.





Rysunek 5.6. Replikacja z wieloma liderami między różnymi centrami danych

### Odporność na problemy z siecią

Ruch między centrami danych jest zwykle przesyłany publicznym internetem, który może być mniej niezawodny niż sieć lokalna w ramach centrum danych. Konfiguracja z jednym liderem jest bardzo podatna na problemy z połączeniami między centrami danych, ponieważ zapis odbywa się synchronicznie z użyciem tego połączenia. W konfiguracji z wieloma liderami z replikacją asynchroniczną problemy z siecią są zwykle lepiej tolerowane. Tymczasowe zakłócenia w sieci nie uniemożliwiają przetwarzania zapisów.

Niektóre bazy domyślnie obsługują konfigurację z wieloma liderami, jednak często jest ona implementowana za pomocą zewnętrznych narzędzi takich jak Tungsten Replicator dla baz MySQL [26], BDR dla baz PostgreSQL [27] i GoldenGate dla baz Oracle [19].

Choć replikacja z wieloma liderami ma zalety, ma też poważną wadę: te same dane mogą zostać jednocześnie zmodyfikowane w dwóch różnych centrach, co wymaga rozwiązania konfliktów (symbolizuje to pole „rozwiązywanie konfliktów” na rysunku 5.6). To zagadnienie omówiono w punkcie „Radzenie sobie z konfliktami przy zapisie”.

Ponieważ w licznych bazach replikacja z wieloma liderami to funkcja wprowadzana jako dodatek, często występują subtelne pułapki związane z konfiguracją i zaskakujące interakcje z innymi funkcjami bazy. Problemem mogą być np. automatycznie inkrementowane klucze, wyzwalacze i więzy integralności. Z tego powodu replikacja z wieloma liderami jest często uznawana za niebezpieczne rozwiązanie, którego w miarę możliwości należy unikać [28].

### Klienci działające w trybie offline

Inna sytuacja, gdzie odpowiednia jest replikacja z wieloma liderami, dotyczy aplikacji, która musi działać także po odłączeniu od internetu.

Wyobraź sobie np. aplikację kalendarza w telefonie komórkowym, laptopie i innych urządzeniach. Potrzebujesz możliwości wyświetlenia spotkań (zgłaszania żądań odczytu) i wprowadzania nowych zdarzeń (zgłaszania żądań zapisu) w dowolnym momencie, niezależnie od tego, czy urządzenie ma akurat połączenie do internetu. Jeśli wprowadzisz zmiany w trybie offline, muszą one zostać zsynchronizowane z serwerem i innymi urządzeniami, gdy dany sprzęt nawiąże połączenie z internetem.

W tym scenariuszu każde urządzenie obejmuje lokalną bazę, która działa jak lider (akceptuje żądania zapisu). Występuje też proces asynchronicznej replikacji z wieloma liderami (synchronizacji) dotyczący replik kalendarza we wszystkich urządzeniach. Opóźnienie replikacji może wynosić godziny, a nawet dni. Zależy to od tego, kiedy urządzenie zyska dostęp do internetu.

Jeśli chodzi o architekturę, ta konfiguracja przypomina skrajną wersję replikacji z wieloma liderami między centrami danych. Każde urządzenie jest tu „centrum danych”, a połączenie sieciowe między nimi jest bardzo niestabilne. Liczne przykłady nieprawidłowych implementacji synchronizacji kalendarza wskazują na to, że trudno uzyskać prawidłową replikację z wieloma liderami.

Istnieją narzędzia, które mają ułatwiać konfigurację rozwiązań z wieloma liderami. Do działania w tym trybie zaprojektowana została np. baza CouchDB [29].

## Edycja zespołowa

Aplikacje do *edycji zespołowej w czasie rzeczywistym* umożliwiają kilku osobom jednoczesne edytowanie dokumentu. Na przykład Etherpad [30] i Google Docs [31] pozwalają wielu użytkownikom na jednoczesną edycję dokumentu tekstowego lub arkusza kalkulacyjnego (używany do tego algorytm pokrótce opisano w punkcie „Automatyczne rozwiązywanie konfliktów”).

Edycja zespołowa zwykle nie jest traktowana jako problem dotyczący replikacji bazy danych. Zadanie to ma jednak wiele wspólnego z wcześniej opisanymi zmianami w trybie offline. Gdy jeden użytkownik edytuje dokument, zmiany są natychmiast wprowadzane w lokalnej replice (w stanie dokumentu w przeglądarce lub aplikacji klienckiej) oraz asynchronicznie replikowane na serwerze i u innych użytkowników, którzy pracują nad tym samym dokumentem.

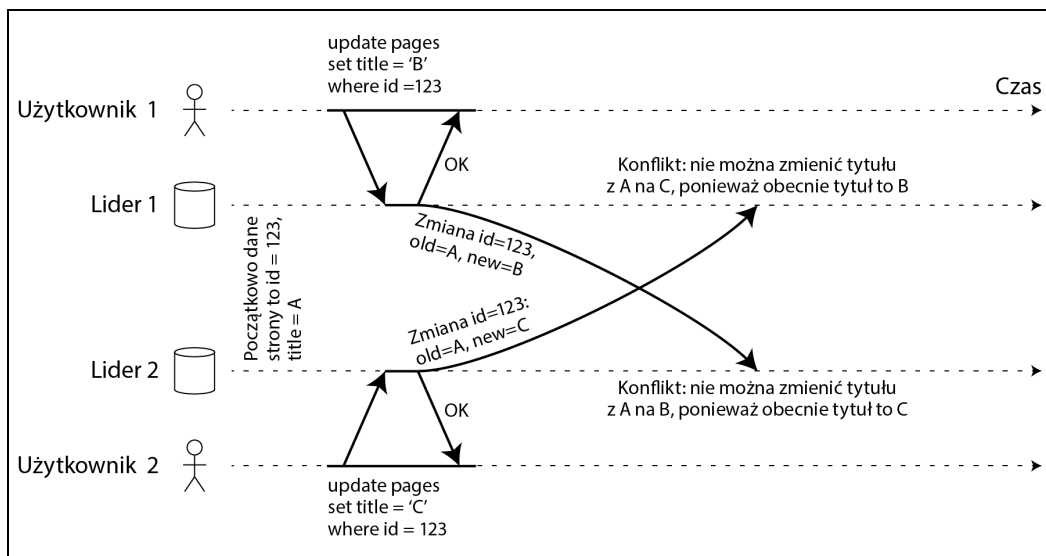
Jeśli potrzebujesz gwarancji, że konflikty w trakcie edycji nie wystąpią, aplikacja musi blokować dokument, zanim użytkownik będzie mógł go zmodyfikować. Jeżeli inny użytkownik zechce edytować ten sam dokument, najpierw będzie musiał poczekać na wprowadzenie zmian i zwolnienie blokady przez pierwszego użytkownika. Ten model współpracy jest odpowiednikiem replikacji z jednym liderem z transakcjami wykonywanymi u lidera.

Jednak aby przyspieszyć współpracę, można zastosować bardzo małą jednostkę zmian (np. wciśnięcie pojedynczych klawiszy) i uniknąć blokad. To podejście pozwala wielu użytkownikom jednocześnie wprowadzać zmiany, ale też rodzi problemy typowe dla replikacji z wieloma liderami, w tym wymaga rozwiązywania konfliktów [32].

## Radzenie sobie z konfliktami przy zapisie

Największy problem w replikacji z wieloma liderami stanowi możliwość wystąpienia konfliktów przy zapisie. Oznacza to, że niezbędne jest rozwiązywanie konfliktów.

Wyobraź sobie np. stronę wiki jednocześnie edytowaną przez dwóch użytkowników, co ilustruje rysunek 5.7. Użytkownik 1 zmienia tytuł strony z A na B, a użytkownik 2 w tym samym czasie zmienia ten tytuł z A na C. Zmiany każdego użytkownika są z powodzeniem wprowadzane u lokalnego lidera. Jednak na etapie asynchronicznej replikacji zmian wykryty zostaje konflikt [33]. W bazie z jednym liderem ten problem nie występuje.



Rysunek 5.7. Konflikt przy zapisie spowodowany przez jednoczesną aktualizację tego samego rekordu przez dwóch liderów

### Synchroniczne i asynchroniczne wykrywanie konfliktów

W bazie z jednym liderem albo druga jednostka zapisująca zostaje zablokowana i oczekuje na zakończenie pierwszego zapisu, albo druga transakcja zapisu jest odrzucana, co zmusza użytkownika do ponowienia próby. Natomiast w konfiguracji z wieloma liderami oba zapisy kończą się powodzeniem, a konflikt jest wykrywany asynchronicznie na późniejszym etapie. Może być wtedy za późno na to, by zażądać od użytkownika rozwiązania konfliktu.

W zasadzie można zastosować synchroniczne wykrywanie konfliktów, czyli oczekiwać na replikację zapisu we wszystkich replikach przed poinformowaniem użytkownika, że zapis zakończył się powodzeniem. Jednak to rozwiązanie skutkuje utratą głównej zalety replikacji z wieloma liderami: umożliwianiu każdej repliki niezależnego akceptowania zapisów. Jeśli odpowiada Ci synchroniczne rozwiązywanie konfliktów, możesz równie dobrze zastosować replikację z jednym liderem.

### Unikanie konfliktów

Najprostsza strategia radzenia sobie z konfliktami polega na ich unikaniu. Jeśli aplikacja może zagwarantować, że wszystkie zapisy danego rekordu będą trafiały do tego samego lidera, konflikty nie wystąpią. Ponieważ liczne implementacje replikacji z wieloma liderami słabo radzą sobie z konfliktami, unikanie konfliktów jest często zalecanym podejściem [34].

Na przykład w aplikacji, gdzie użytkownik może edytować własne dane, da się zagwarantować, że żądania od konkretnego użytkownika będą zawsze kierowane do tego samego centrum danych oraz że odczyty i zapisy będą obsługiwane przez lidera z tego centrum. Poszczególnym użytkownikom można przypisać różne „domowe” centra danych (np. na podstawie geograficznej bliskości do użytkownika), jednak z perspektywy pojedynczych użytkowników system działa tak jak w konfiguracji z jednym liderem.

Jednak czasem pożądana jest zmiana ustalonego dla rekordu lidera — np. z powodu awarii jednego centrum danych i konieczności przekierowania ruchu do innego centrum albo dlatego, że użytkownik przeniósł się i obecnie znajduje się bliżej innego centrum. W takich sytuacjach unikanie konfliktów przestaje działać i trzeba uwzględnić możliwość równoczesnych zapisów u różnych liderów.

### Zbieżne dochodzenie do spójnego stanu

Baza z jednym liderem wprowadza zapisy sekwencyjnie. Jeśli to samo pole jest aktualizowane kilkakrotnie, ostatni zapis określa ostateczną wartość tego pola.

W konfiguracji z wieloma liderami nie występuje zdefiniowana kolejność zapisów, dlatego nie jest jasne, jaka powinna być ostateczna wartość. Na rysunku 5.7 u lidera 1 tytuł najpierw jest aktualizowany do wartości B, a następnie do wartości C. U lidera 2 najpierw następuje aktualizacja do wartości C, a następnie do wartości B. Żadna kolejność nie jest „bardziej poprawna” od innej.

Gdyby każda replika wprowadzała zapisy w kolejności ich odnotowania, stan bazy byłby niespójny. Ostateczną wartością byłoby C u lidera 1 i B u lidera 2. To nieakceptowalne. W każdym systemie replikacji trzeba gwarantować, że dane we wszystkich replikach ostatecznie będą identyczne. Dlatego baza musi rozwiązywać konflikty w *zbieżny* sposób, co oznacza, że po zreplikowaniu wszystkich zmian wszystkie repliki muszą dojść do tej samej wartości końcowej.

Istnieją różne sposoby zbieżnego rozwiązywania konfliktów:

- Każdemu zapisowi przypisz unikatowy identyfikator (np. znacznik czasu, długą liczbę losową, identyfikator UUID lub skrót klucza i wartości), wybierz zapis o najwyższym identyfikatorze jako *zwycięzcę* i odrzuć pozostałe zapisy. Gdy używa się znacznika czasu, ta technika nosi nazwę LWW (ang. *last write wins*, czyli ostatni zapis wygrywa). Choć to popularne podejście, grozi ono utratą danych [35]. Szczegółowe omówienie tego rozwiązania znajdziesz w końcowej części tego rozdziału („Wykrywanie jednoczesnych zapisów”).
- Przypisz każdej replice unikatowy identyfikator i spraw, by zapisy z repliki o najwyższym numerze zawsze były traktowane priorytetowo względem zapisów z replik o niższych numerach. To podejście także grozi utratą danych.
- Scalaj wartości ze sobą. Możesz np. porządkować je alfabetycznie, a następnie złączać (na rysunku 5.7 scalony tytuł może mieć postać „B/C”).
- Zapisz konflikt w odrębnej strukturze danych, która zachowuje wszystkie informacje, i napisz w aplikacji kod eliminujący konflikty w późniejszym czasie (np. zadając pytanie użytkownikowi).

## Niestandardowy kod do rozwiązywania konfliktów

To, który sposób rozwiązywania konfliktów będzie najbardziej odpowiedni, może zależeć od aplikacji. Dlatego większość narzędzi do replikacji w modelu z wieloma liderami umożliwia dodanie logiki rozwiązywania konfliktów w kodzie aplikacji. Ten kod może być wykonywany przy zapisie lub przy odczycie:

### *Przy zapisie*

Gdy tylko system bazodanowy wykryje konflikt w dzienniku replikowanych zmian, wywołuje mechanizm obsługi konfliktów. Na przykład Bucardo umożliwia napisanie w tym celu fragmentu kodu w Perlu. Mechanizm tego rodzaju zwykle nie może wyświetlić pytania użytkownikowi, ponieważ działa w procesie tła i musi szybko wykonać zadanie.

### *Przy odczycie*

W momencie wykrycia konfliktu zachowywane są wszystkie sprzeczne zapisy. Gdy dane są ponownie wczytywane, do aplikacji zwracane są różne wersje danych. Aplikacja może wyświetlić pytanie użytkownikowi lub automatycznie rozwiązać konflikt i zapisać wynik z powrotem w bazie. Tak działa np. baza CouchDB.

Zauważ, że rozwiązywanie konfliktów jest zwykle stosowane na poziomie poszczególnych wierszy lub dokumentów, a nie całej transakcji [36]. Dlatego jeśli transakcja atomowo wprowadza kilka różnych zapisów (zob. rozdział 7.), w kontekście rozwiązywania konfliktów każdy z nich jest traktowany jako odrębny.

## Czym jest konflikt?

Niektóre rodzaje konfliktów są oczywiste. W przykładzie z rysunku 5.7 dwa zapisy jednocześnie zmodyfikowały to samo pole tego samego rekordu, ustawiając je na dwie różne wartości. Nie ma wątpliwości, że zachodzi tu konflikt.

Wykrycie konfliktów innego rodzaju może być trudniejsze. Pomyśl np. o systemie rezerwowania sal konferencyjnych. Taki system śledzi, która sala została zarezerwowana przez daną grupę osób na określony czas. Aplikacja musi gwarantować, że każda sala może zostać zarezerwowana na dany czas tylko przez jedną grupę. Oznacza to, że nachodzące na siebie rezerwacje tej samej sali nie są dopuszczalne. W tym scenariuszu konflikt może wystąpić, jeśli pojawią się dwie różne rezerwacje tego samego pomieszczenia na ten sam czas. Nawet jeśli aplikacja sprawdza dostępność przed pozwoleniem użytkownikowi na dokonanie rezerwacji, może wystąpić konflikt, jeśli sala zostanie zarezerwowana u dwóch różnych liderów.

Nie istnieje proste gotowe rozwiązanie tego problemu, jednak dalsze rozdziały pozwolą Ci dobrze go zrozumieć. Więcej przykładów konfliktów zawiera rozdział 7., a w rozdziale 12. opisane są skalowalne sposoby wykrywania i rozwiązywania konfliktów w systemach z replikacją.

## Automatyczne rozwiązywanie konfliktów

Reguły rozwiązywania konfliktów szybko mogą się stać skomplikowane, a niestandardowy kod bywa podatny na błędy. Często podawany przykład zaskakujących efektów spowodowanych mechanizmem rozwiązywania konfliktów dotyczy Amazonu. Przez pewien czas logika rozwiązywania konfliktów w koszyku zakupów zachowywała produkty dodane do koszyka, ale nie zawsze zachowywała operacje usuwania z niego towarów. Dlatego klienci czasem widzieli w koszykach produkty, które wcześniej usunęli [37].

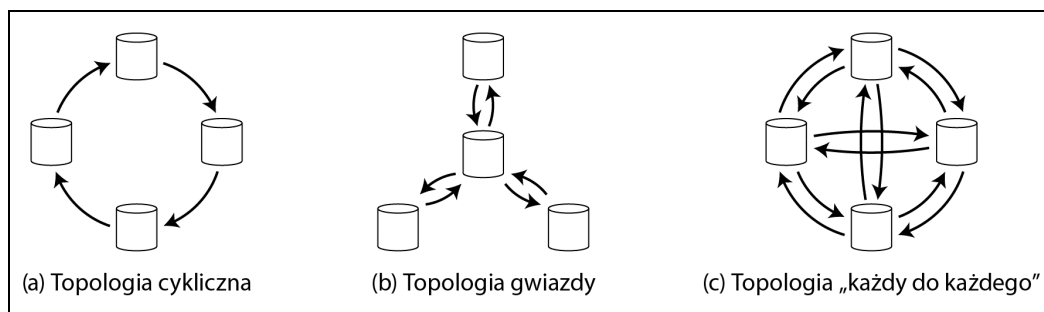
Przeprowadzono ciekawe badania nad automatycznym rozwiązywaniem konfliktów spowodowanych jednoczesnymi modyfikacjami danych. Warto wspomnieć o kilku obszarach takich badań:

- **Typy CRDT** (ang. *conflict-free replicated datatypes*) [32, 38] to rodzina struktur danych obejmujących zbiory, odwzorowania, listy uporządkowane, liczniki itd., które mogą być jednocześnie edytowane przez wielu użytkowników i które automatycznie eliminują konflikty w sensowny sposób. Niektóre typy CRDT zostały zaimplementowane w bazie Riak 2.0 [39, 40].
- **Trwałe struktury danych z możliwością scalania** [41] bezpośrednio śledzą historię zmian (podobnie jak robi to system kontroli wersji Git) i używają funkcji scalania trójstronnego (w typach CRDT używane jest scalanie dwustronne).
- **Mechanizm OT** (ang. *operational transformation*) [42] to algorytm rozwiązywania konfliktów używany w aplikacjach do edycji zespołowej takich jak Etherpad [30] i Google Docs [31]. Został zaprojektowany specjalnie z myślą o jednoczesnej edycji uporządkowanych list elementów, np. list znaków w dokumencie tekstowym.

Implementacje tych algorytmów w bazach są wciąż na początkowym etapie, jednak w przyszłości zostaną prawdopodobnie zintegrowane z systemami danych z replikacją. Automatyczne rozwiązywanie konfliktów może sprawić, że synchronizacja danych w modelu z wieloma liderami będzie znacznie łatwiejsza w obsłudze dla aplikacji.

## Topologie replikacji z wieloma liderami

*Topologia replikacji* opisuje ścieżki komunikacji, którymi zapisy są przekazywane między węzłami. Jeśli występuje dwóch liderów (tak jak na rysunku 5.7), istnieje tylko jedna sensowna topologia: lider 1 musi przysyłać wszystkie zapisy do lidera 2 i na odwrót. Gdy liderów jest więcej niż dwóch, możliwe są różne topologie. Niektóre z nich pokazano na rysunku 5.8.



Rysunek 5.8. Trzy przykładowe topologie replikacji z wieloma liderami

Najbardziej ogólna topologia to *każdy do każdego* („c” na rysunku 5.8), w której każdy lider przekazuje zapisy do wszystkich pozostałych liderów. Używane są też jednak topologie z pewnymi ograniczeniami. Na przykład MySQL domyślnie obsługuje tylko *topologię cykliczną* [34], w której każdy węzeł otrzymuje zapisy od jednego węzła i przekazuje je (razem z własnymi zapisami) do jednego innego. Inna popularna topologia ma kształt *gwiazdy*<sup>4</sup>. Jeden wyznaczony węzeł główny przekazuje zapisy do wszystkich pozostałych węzłów. Topologię gwiazdy można uogólnić do postaci drzewa.

W topologiach cyklicznej i gwiazdy zapis musi czasem przejść przez kilka węzłów, zanim dotrze do wszystkich replik. Dlatego węzły muszą przekazywać zmiany danych otrzymane od innych węzłów. Aby zapobiec powstawaniu nieskończonych pętli replikacji, każdy węzeł otrzymuje unikatowy identyfikator, a w dzienniku replikacji każdy zapis jest opatrzony identyfikatorami wszystkich węzłów, przez które przechodził [43]. Gdy węzeł otrzymuje zmianę danych opatrzoną własnym identyfikatorem, ignoruje ją, ponieważ wie, że przetworzył już tę zmianę.

Problem z topologiami cykliczną i gwiazdy polega na tym, że jeśli zawiedzie choć jeden węzeł, może to zakłócić przepływ związanych z replikacją komunikatów między innymi węzłami i uniemożliwić komunikację do czasu naprawienia niesprawnego węzła. Topologię można zrekonfigurować, aby pominąć uszkodzony węzeł, jednak w większości instalacji taką rekonfigurację trzeba przeprowadzać ręcznie. Odporność na błędy gęściej połączonych topologii (takich jak „każdy do każdego”) jest wyższa, ponieważ komunikaty mogą przepływać różnymi ścieżkami, co pozwala uniknąć powstania jednego punktu podatności na awarie.

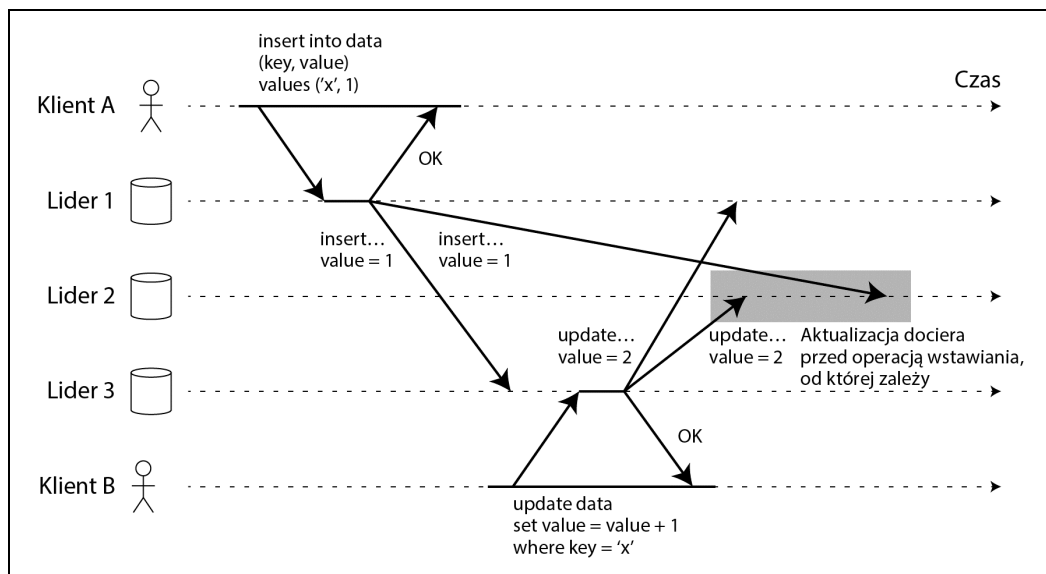
Jednak w topologiach „każdy do każdego” też mogą występować problemy. Przede wszystkim niektóre połączenia sieciowe mogą działać szybciej niż inne (np. z powodu przeciążenia sieci), co skutkuje tym, że niektóre komunikaty dotyczące replikacji „wyprzedzają” inne. Ilustruje to rysunek 5.9.

Na rysunku 5.9 klient A wstawia wiersz do tabeli u lidera 1, a klient B aktualizuje ten wiersz u lidera 3. Jednak lider 2 może otrzymać zapisy w innej kolejności: najpierw aktualizację (która, z perspektywy tego lidera, dotyczy wiersza nieistniejącego w bazie), a dopiero potem powiązaną operację wstawiania (która powinna nastąpić przed aktualizacją).

To problem przyczynowości podobny do opisanego w punkcie „Odczyty ze spójnym przedrostkiem”. Aktualizacja zależy od wcześniejszej operacji wstawiania, dlatego trzeba się upewnić, że wszystkie węzły najpierw wstawią dane, a potem je zaktualizują. Samo dołączenie znacznika czasu do każdego zapisu nie wystarczy, ponieważ nie można ufać temu, że zegary będą wystarczająco zsynchronizowane, aby poprawnie uporządkować zdarzenia u lidera 2 (zob. rozdział 8.).

Aby zapewnić prawidłową kolejność zdarzeń, można zastosować technikę *wektorów wersji* opisaną w dalszej części rozdziału (zob. punkt „Wykrywanie jednoczesnych zapisów”). Jednak w licznych systemach replikacji z wieloma liderami techniki wykrywania konfliktów są słabo zaimplementowane. Na przykład w czasie, gdy powstaje ta książka, BDR w bazach PostgreSQL nie zapewnia przyczynowego uporządkowania zapisów [27], a Tungsten Replicator w bazach MySQL nie próbuje nawet wykrywać konfliktów [34].

<sup>4</sup> Nie należy mylić tej topologii ze *schematem gwiazdy* (zob. punkt „Gwiazdy i płatki śniegu — schematy używane w analityce”) opisującym strukturę modelu danych, a nie topologię komunikacji między węzłami.



Rysunek 5.9. W replikacji z wieloma liderami zapisy w niektórych replikach mogą się pojawiać w niewłaściwej kolejności

Jeśli używasz systemu z replikacją z wieloma liderami, warto mieć świadomość opisanych problemów, starannie przeczytać dokumentację oraz dokładnie przetestować bazę danych, aby mieć pewność, że rzeczywiście zapewnia zakładane gwarancje.

## Replikacja bez lidera

Omawiane wcześniej w rozdziale sposoby replikacji (z jednym liderem i z wieloma liderami) są oparte na tym, że klient przesyła żądanie zapisu do jednego węzła (lidera), a system bazodanowy kopiuje ten zapis do innych replik. Lider określa kolejność, w jakiej zapisy powinny być przetwarzane, a obserwatorzy wprowadzają zapisy lidera w tej samej kolejności.

W niektórych systemach składowania danych stosowane jest inne podejście. Zrezygnowano w nich z lidera, a każda replika może bezpośrednio akceptować zapisy od klientów. Niektóre z pierwszych systemów danych z replikacją nie obejmowały lidera [1, 44], jednak w epoce dominacji baz relacyjnych to podejście zostało prawie zapomniane. Ponownie stało się modną architekturą baz danych po tym, jak Amazon zastosował je do wewnętrznego systemu *Dynamo* [37]<sup>5</sup>. Riak, Cassandra i Voldemort to otwarte bazy danych z modelami replikacji bez lidera inspirowanymi systemem *Dynamo*, dlatego rozwiązania tego rodzaju są też nazywane *bazami w stylu Dynamo*.

W niektórych implementacjach bez lidera klient bezpośrednio przesyła zapisy do kilku replik. W innych robi to koordynator na rzecz klienta. Jednak koordynator, w odróżnieniu od lidera, nie wymusza określonej kolejności zapisów. Jak się przekonasz, ta różnica w projekcie ma istotny wpływ na sposób używania bazy.

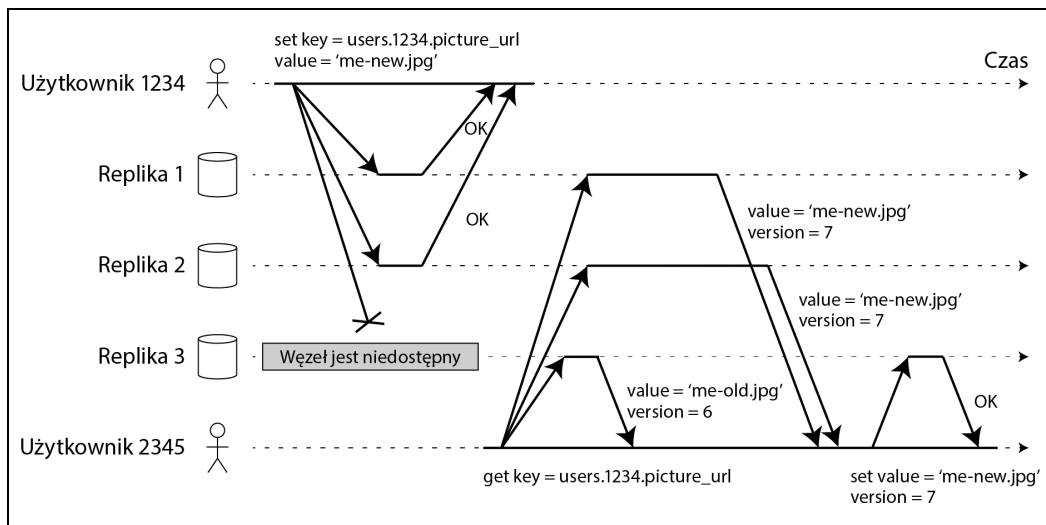
<sup>5</sup> *Dynamo* nie jest dostępne dla użytkowników poza usługami Amazonu. Mylące jest to, że usługi AWS udostępniają hostowaną bazę *DynamoDB* o zupełnie odmiennej architekturze (opartej na replikacji z jednym liderem).



## Zapis danych w bazie, gdy węzeł nie działa

Wyobraź sobie, że masz bazę z trzema replikami i jedna z nich jest aktualnie niedostępna — możliwe, że jest restartowana w celu zainstalowania aktualizacji systemu. Jeśli w konfiguracji opartej na liderze chcesz kontynuować przetwarzanie zapisów, możliwe, że potrzebne będzie przełączanie awaryjne (zob. punkt „Obsługa przestojów węzłów”).

W konfiguracji bez lidera przełączanie awaryjne nie jest stosowane. Na rysunku 5.10 pokazano, co się dzieje w takim modelu. Klient (użytkownik 1234) równoległe przesyła zapis do wszystkich trzech replik. Dwie dostępne repliki akceptują dane, natomiast niedostępna replika go nie odbiera. Załóżmy, że wystarczy, aby dwie z trzech replik zatwierdziły zapis. Gdy użytkownik 1234 otrzyma dwie odpowiedzi OK, zapis uznaje się za udany. Klient ignoruje wtedy fakt, że jedna z replik nie zapisała danych.



Rysunek 5.10. Zapis i odczyt na podstawie kworum oraz uspoźnianie przy odczycie po przestoju węzła

Teraz wyobraź sobie, że niedostępny węzeł znów stanie się dostępny i klient zacznie wczytywać z niego dane. Wszystkie zapisy, które nastąpiły w czasie, gdy węzeł nie działał, nie zostały w nim wprowadzone. Dlatego jeśli wczytujesz dane z tego węzła, możesz otrzymać *przestarzałe* wartości.

Aby rozwiązać ten problem, klient w momencie odczytu danych z bazy nie przesyła żądania do tylko jednej repliki. Zamiast tego *żądania odczytu są przysyłane równoległe do kilku węzłów*. Klient może otrzymać od poszczególnych węzłów różne odpowiedzi — np. aktualną wartość z jednego węzła i nieaktualną z innego. Do określania, która wartość jest nowsza, służą numery wersji (zob. punkt „Wykrywanie jednoczesnych zapisów”).

### Uspójnianie przy odczycie i eliminowanie entropii

Stosowany schemat replikacji powinien gwarantować, że ostatecznie wszystkie dane zostaną skopiowane do każdej repliki. Jak niedostępny węzeł uzupełnia pominięte zapisy, gdy wznowi pracę?

W bazach w stylu Dynamo często używane są dwa mechanizmy:

### *Uspójnianie przy odczycie*

Gdy klient równolegle wczytuje dane z kilku węzłów, może wykryć nieaktualne odpowiedzi. Na przykład na rysunku 5.10 użytkownik 2345 otrzymuje wartość w wersji 6 z repliki 3 i wartość w wersji 7 z replik 1 i 2. Klient wykrywa, że w replice 3 znajduje się przestarzała wartość, i zapisuje w tej replice nowszą wartość. To podejście sprawdza się dobrze w przypadku często wczytywanych wartości.

### *Proces eliminowania entropii*

Dodatkowo w niektórych bazach używany jest działający w tle proces, który stale szuka różnic w danych między replikami i kopiuje brakujące dane z jednej repliki do innej. W odróżnieniu od dziennika replikacji w modelu opartym na liderze *proces eliminowania entropii* nie kopiuje zapisów w konkretnej kolejności. Ponadto kopiowanie danych może się odbywać ze znacznym opóźnieniem.

Nie we wszystkich systemach zaimplementowane są oba te mechanizmy. Na przykład w bazie Voldemort obecnie nie stosuje się procesu eliminowania entropii. Zauważ, że brak takiego procesu sprawia, iż niektóre repliki mogą nie obejmować rzadko wczytywanych wartości, co zmniejsza trwałość danych, ponieważ uspójnianie przy odczycie jest wykonywane tylko w momencie wczytywania danych przez aplikację.

### **Odczyty i zapisy zgodne z kworum**

Na przykładzie z listingu 5.10 zapis był uznawany za udany nawet wtedy, gdy przetworzyły go tylko dwie z trzech replik. Co się stanie, jeśli tylko jedna z trzech replik zaakceptuje zapis? Jak daleko można się posunąć w tym podejściu?

Jeśli wiadomo, że każdy udany zapis na pewno zostaje wprowadzony w przynajmniej dwóch z trzech replik, oznacza to, iż najwyżej jedna replika może być nieaktualna. Dlatego wczytanie danych z przynajmniej dwóch replik gwarantuje, że co najmniej jedna z nich będzie aktualna. Jeżeli trzecia replika nie działa lub odpowiada z opóźnieniem, odczyt i tak pozwala uzyskać aktualną wartość.

Ogólnie jeśli jest  $n$  replik, każdy zapis musi zostać zatwierdzony przez  $w$  węzłów, aby móc go uznać za udany, a przy każdym odczycie trzeba przesłać zapytanie do przynajmniej  $r$  węzłów. W tym przykładzie  $n = 3$ ,  $w = 2$  i  $r = 2$ . Dopóki  $w + r > n$ , można oczekiwać uzyskania aktualnej wartości w momencie odczytu, ponieważ przynajmniej jeden z  $r$  węzłów używanych przy odczycie musi być aktualny. Odczyty i zapisy zgodne z wartościami  $r$  i  $w$  to odczyty i zapisy zgodne z *kworum* [44]<sup>6</sup>. Wartości  $r$  i  $w$  możesz traktować jak minimalną liczbę głosów potrzebnych do tego, by odczyty i zapisy zostały uznane za prawidłowe.

W bazach w stylu Dynamo parametry  $n$ ,  $w$  i  $r$  można zwykle konfigurować. Częstym wyborem jest używanie jako  $n$  liczby nieparzystej (zwykle 3 lub 5) i ustawianie  $w = r = (n + 1) / 2$  (z zaokrągleniem). Możesz jednak dowolnie dostosować te wartości. Na przykład przy obciążeniu roboczym z niewielką liczbą zapisów i wieloma odczytami odpowiednie mogą być wartości  $w = n$  i  $r = 1$ .

<sup>6</sup> Kworum tego rodzaju można nazwać *ściśłym* w odróżnieniu od *kworum niepełnego* (opisanego w punkcie „Kwora niepełne i mechanizm hinted-handoff”).

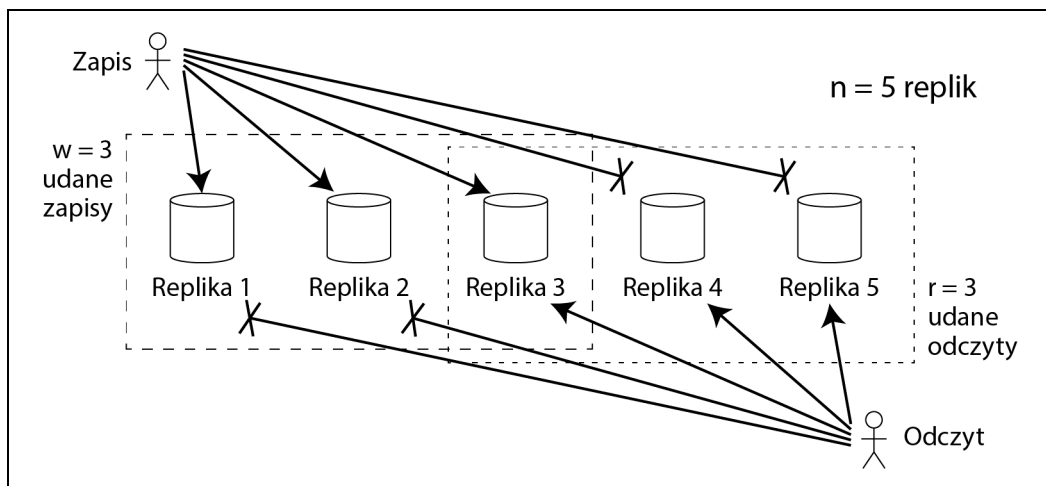
Dzięki temu odczyty są szybsze, jednak wadę tego rozwiązania stanowi to, że już jeden uszkodzony węzeł powoduje niepowodzenie wszystkich zapisów w bazie.



W klastrze może się znajdować więcej niż  $n$  węzłów, jednak dana wartość jest przechowywana tylko w  $n$  węzłach. Dzięki temu zbiór danych można podzielić, zapewniając w ten sposób obsługę zbiorów danych większych, niż może pomieścić jeden węzeł. Do podziału danych na partycje wrócimy w rozdziale 6.

Warunek określający kworum,  $w + r > n$ , pozwala systemowi tolerować niedostępność węzłów w następujący sposób:

- Jeśli  $w < n$ , nadal można przetwarzać zapisy, gdy węzeł jest niedostępny.
- Jeżeli  $r < n$ , nadal można przetwarzać odczyty, gdy węzeł jest niedostępny.
- Dla  $n = 3$ ,  $w = 2$  i  $r = 2$  tolerowany jest jeden niedostępny węzeł.
- Dla  $n = 5$ ,  $w = 3$  i  $r = 3$  tolerowane są dwa niedostępne węzły. Tę sytuację przedstawiono na rysunku 5.11.
- Standardowo odczyty i zapisy zawsze są przesyłane równolegle do wszystkich  $n$  replik. Parametry  $w$  i  $r$  określają, na ile węzłów należy czekać (tzn. ile z  $n$  węzłów musi poinformować o powodzeniu przed uznaniem, że odczyt lub zapis był udany).



Rysunek 5.11. Gdy  $w + r > n$ , przynajmniej jedna z  $r$  replik używanych przy odczycie musiała otrzymać najnowszy udany zapis

Jeśli dostępnych jest mniej niż wymagane  $w$  lub  $r$  węzłów, zapisy lub odczyty zwracają błąd. Węzeł może być niedostępny z wielu powodów: ponieważ nie działa (uległ awarii lub odcięto zasilanie), z powodu błędu w wykonywaniu operacji (nie można dokonać zapisu, ponieważ dysk jest pełny), z uwagi na zakłócenie pracy sieci między klientem a węzłem albo z wielu innych przyczyn. Ważne jest tylko to, czy węzeł zwrócił poprawną odpowiedź. Nie trzeba uwzględniać różnych rodzajów awarii.

## Ograniczenia określania spójności na podstawie kworum

Jeśli jest  $n$  replik i wybrałeś takie  $w$  i  $r$ , że  $w + r > n$ , ogólnie możesz oczekiwać, że każdy odczyt zwróci najnowszą wartość zapisaną dla danego klucza. Dzieje się tak, ponieważ wtedy zbiór węzłów, w których zapisano dane, i zbiór węzłów zwracających informacje zawsze się pokrywają. Oznacza to, że wśród węzłów używanych do odczytu musi się znajdować przynajmniej jeden z najnowszą wartością (ilustruje to rysunek 5.11).

Często  $r$  i  $w$  są tak dobierane, by stanowić większość węzłów (więcej niż  $n/2$ ), ponieważ to gwarantuje, że  $w + r > n$  nawet przy awarii  $n/2$  węzłów. Jednak kworum nie musi być większością. Ważne jest tylko to, by zbiory węzłów używanych w operacjach odczytu i zapisu obejmowały choć jeden wspólny węzeł. Możliwe są też inne układy kworum, co pozwala na pewną swobodę w trakcie projektowania algorytmów rozproszonych [45].

Możesz też użyć mniejszych wartości  $w$  i  $r$ , tak by  $w + r \leq n$  (warunek uzyskania kworum nie zostaje wtedy spełniony). Wtedy odczyty i zapisy nadal będą przesyłane do  $n$  węzłów, jednak wymagana jest mniejsza liczba udanych odpowiedzi, aby operacja zakończyła się powodzeniem.

Mniejsze wartości  $w$  i  $r$  zwiększają ryzyko odczytu przestarzałych wartości, ponieważ istnieje większe prawdopodobieństwo, że odczyt nie obejmował węzła z najnowszą wartością. Plus stanowi to, że ta konfiguracja pozwala uzyskać niższe opóźnienie i wyższą dostępność. Jeśli w sieci występują zakłócenia i wiele replik staje się niedostępnych, istnieje duża szansa, że możliwe będzie kontynuowanie przetwarzania odczytów i zapisów. Dopiero gdy liczba dostępnych replik spadnie poniżej  $w$  lub  $r$ , baza stanie się niedostępna do zapisu lub odczytu.

Jednak nawet przy  $w + r > n$  zdarzają się przypadki brzegowe, w których zwracane są nieaktualne wartości. Takie sytuacje zależą od implementacji. Oto niektóre możliwe scenariusze:

- Jeśli używane jest kworum niepełne (ang. *sloppy quorum*; zob. punkt „Kwora niepełne i mechanizm hinted-handoff”), w zapisów może trafić do innych węzłów niż te, które są używane do  $r$  odczytów. W takiej sytuacji nie ma gwarancji pokrywania się  $r$  węzłów z  $w$  węzłami [46].
- Jeśli dwa zapisy są wprowadzane jednocześnie, nie jest jasne, który z nich wystąpił jako pierwszy. W takiej sytuacji jedynym bezpiecznym rozwiązaniem będzie scalanie jednocześnie wprowadzonych zapisów (zob. punkt „Radzenie sobie z konfliktami przy zapisie”). Gdy zwycięzca jest wybierany na podstawie znacznika czasu (ostatni zapis wygrywa), zapisy mogą zostać utracone z powodu odchyień zegara [35]. Do tego tematu wracamy w punkcie „Wykrywanie jednoczesnych zapisów”.
- Jeżeli zapis odbywa się równolegle z odczytem, zapis może zostać odzwierciedlony tylko w niektórych replikach. W takim scenariuszu nie wiadomo, czy odczyt zwróci starą, czy nową wartość.
- Jeśli zapis zakończył się sukcesem w niektórych replikach, ale niepowodzeniem w innych (np. z powodu zapełnienia dysków w niektórych węzłach), i łącznie powiódł się w mniej niż  $w$  replikach, nie jest wycofywany w replikach, w których się udał. To oznacza, że jeśli zapis został zarejestrowany jako nieudany, kolejne odczyty mogą zwrócić wartość z tego zapisu, ale nie jest to pewne [47].

- Jeżeli węzeł obejmujący nową wartość ulegnie awarii, a dane w nim zostaną przywrócone z użyciem repliki zawierającej dawną wartość, liczba replik przechowujących nową wartość może spaść poniżej  $w$ , co narusza warunek zapewnienia kworum.
- Nawet jeśli wszystko inne działa poprawnie, występują przypadki brzegowe, w których układ zdarzeń w czasie jest nieszczęśliwy. Opisano to w punkcie „Liniiowość i kwora”.

Dlatego choć kwora na pozór gwarantują, że odczyt zwraca najnowszą zapisaną wartość, w praktyce to nie takie proste. Bazy w stylu Dynamo są zwykle zoptymalizowane pod kątem przypadków użycia, gdzie akceptowalna jest spójność ostateczna. Parametry  $w$  i  $r$  pozwalają dostosować prawdopodobieństwo wczytania nieaktualnych wartości, jednak rozsądnie jest nie traktować uzyskanych gwarancji jako absolutnych.

Przed wszystkim zwykle nie są zapewniane gwarancje opisane w punkcie „Problemy z opóźnieniem replikacji” (odczyt własnych zapisów, odczyty monotoniczne i odczyty ze spójnym przedrostkiem), dlatego w aplikacjach mogą występować wcześniej wymienione anomalie. Mocniejsze gwarancje zwykle wymagają transakcji lub konsensusu. Do tych tematów wrócimy w rozdziałach 7. i 9.

### Monitorowanie nieaktualnych danych

Z perspektywy operacyjnej ważne jest, by monitorować, czy bazy zwracają aktualne wyniki. Nawet jeśli aplikacja toleruje nieaktualne odczyty, trzeba mieć świadomość stanu replikacji. Jeżeli jest ona nadto opóźniona, powinna zgłaszać alert, abyś mógł zbadać przyczynę (może nią być np. problem w sieci lub przeciążony węzeł).

W replikacji opartej na liderze baza zwykle udostępnia wskaźniki dotyczące opóźnienia replikacji. Możesz je przesłać do systemu monitorowania. Jest to możliwe, ponieważ zapisy są stosowane w liderze i obserwatorach w tej samej kolejności, a każdy węzeł przechowuje określoną pozycję w dzienniku replikacji (liczbę wprowadzonych lokalnie zapisów). Odjęcie aktualnej pozycji u obserwatora od aktualnej pozycji u lidera pozwala zmierzyć poziom opóźnienia replikacji.

Jednak w systemach z replikacją bez lidera nie występuje stała kolejność wprowadzania zapisów, dlatego monitorowanie jest trudniejsze. Ponadto jeśli baza stosuje tylko uspojnianie przy odczycie (bez eliminowania entropii), nie ma ograniczenia określającego, jak dawna może być wartość. Jeżeli jest ona wczytywana rzadko, wartość zwracana przez nieaktualną replikę może być bardzo dawna.

Prowadzono badania nad pomiarem nieaktualności replik w bazach z replikacją bez lidera i prognozowaniem oczekiwanego procentu nieaktualnych odczytów w zależności od parametrów  $n$ ,  $w$  i  $r$  [48]. Niestety nie jest to jeszcze powszechna praktyka, jednak dobrze byłoby wprowadzić pomiary nieaktualności w standardowych zestawach wskaźników w bazach. Spójność ostateczna stanowi celowo nieprecyzyjną gwarancję, jednak na potrzeby eksploatacji ważne jest, aby móc ilościowo określić „ostateczność”.

### Kwora niepełne i mechanizm hinted-handoff

Bazy z odpowiednio skonfigurowanymi kworami mogą tolerować awarie poszczególnych węzłów bez obaw o przełączanie awaryjne. Są też odporne na spowolnienie poszczególnych węzłów, ponieważ żądania nie muszą oczekiwać na reakcję wszystkich  $n$  węzłów. Dane mogą zostać zwrócone

po odpowiedzi w lub  $r$  węzłów. Te cechy sprawiają, że bazy z replikacją bez lidera są atrakcyjne w scenariuszach, gdy potrzebna jest wysoka dostępność i niskie opóźnienie przy tolerancji na to, że niektóre odczyty będą nieaktualne.

Jednak kwora (w opisaney wcześniej postaci) nie są tak odporne na błędy, jak jest to możliwe. Zakłócenia w sieci mogą łatwo spowodować odcięcie klienta od dużej liczby węzłów bazy. Choć te węzły działają, a inne klienty czasem mogą nawiązać z nimi połączenie, dla odciętego klienta węzły z bazą mogłyby równie dobrze nie działać. W takiej sytuacji prawdopodobne jest, że pozostanie mniej niż  $w$  lub  $r$  dostępnych węzłów, dlatego klient nie może uzyskać kworum.

W dużym klastrze (ze znacznie więcej niż  $n$  węzłów) prawdopodobne jest, że klient w trakcie zakłócenia pracy sieci może się połączyć z *niektórymi* węzłami bazy, tylko nie są to węzły potrzebne do uzyskania kworum dotyczącego konkretnej wartości. W takiej sytuacji projektanci bazy muszą pójść na kompromis:

- Czy lepiej zwracać błędy w odpowiedzi na wszystkie żądania, dla których nie można uzyskać kworum  $w$  lub  $r$  węzłów?
- A może należy akceptować zapisy i umieszczać je w niektórych węzłach, które są dostępne, ale nie należą do  $n$  węzłów, gdzie zwykle dana wartość się znajduje?

To drugie rozwiązanie to *kworum niepełne* [37]. Zapisy i odczyty nadal wymagają wtedy  $w$  i  $r$  udanych odpowiedzi, jednak używane mogą być węzły, które nie należą do wyznaczonej grupy  $n$  „domowych” węzłów dla danej wartości. Oto analogia: jeśli zaciął Ci się zamek w drzwiach, możesz zastukać do sąsiada i zapytać, czy pozwoli Ci tymczasowo skorzystać z jego kanapy.

Po wyeliminowaniu zakłóceń w sieci wszelkie zapisy, które dany węzeł tymczasowo zaakceptował w zamian za inny węzeł, są przesyłane do odpowiednich węzłów „domowych”. Ten mechanizm to *hinted-handoff*. Gdy znajdziesz klucze do swojego mieszkania, Twój sąsiad uprzejmie poprosi Cię, żebyś opuścił jego kanapę i poszedł do siebie.

Kwora niepełne są użyteczne zwłaszcza do zwiększania dostępności zapisów. Dopóki dostępnych jest *dowolnych*  $w$  węzłów, baza może akceptować zapisy. To jednak oznacza, że gdy  $w + r > n$ , nie ma pewności, że wczytana zostanie najnowsza wartość dla danego klucza, ponieważ mogła być ona tymczasowo zapisana w węzłach spoza  $n$  [47].

Dlatego kwora niepełne w ogóle nie są kworami w tradycyjnym sensie. Dają tylko gwarancję trwałości, dotyczącą tego, że dane są składowane gdzieś w  $w$  węzłach. Do czasu zakończenia operacji *hinted-handoff* nie ma gwarancji, że odczyt z  $r$  węzłów pozwoli uzyskać zapisaną wartość.

Kwora niepełne są opcjonalne we wszystkich popularnych implementacjach Dynama. W bazie Riak są one domyślnie włączone, a w bazach Cassandra i Voldemort domyślnie wyłączone [46, 49, 50].

## **Eksploatacja w wielu centrach danych**

Wcześniej opisano replikację między centrami danych jako scenariusz do zastosowania replikacji z wieloma liderami (zob. punkt „Replikacja z wieloma liderami”). Do eksploatacji w wielu centrach danych odpowiednia jest też replikacja bez lidera, ponieważ została tak zaprojektowana, aby tolerowała sprzeczne jednoczesne zapisy, zakłócenia w sieci i skoki opóźnienia.

W bazach Cassandra i Voldemort obsługa wielu centrów danych jest zaimplementowana za pomocą normalnego modelu bez lidera. Liczba replik,  $n$ , obejmuje węzły z wszystkich centrów danych, a w konfiguracji można określić, ile z  $n$  replik ma się znajdować w każdym centrum. Każdy zapis od klienta jest przesyłany do wszystkich replik (niezależnie od centrum danych), jednak klient zwykle czeka tylko na potwierdzenia od kworum węzłów z jego lokalnego centrum danych, dlatego nie odczuwa opóźnień i zakłóceń dotyczących połączenia między centrami. Zapisy o wyższym opóźnieniu przesyłane do innych centrów danych są często skonfigurowane jako asynchroniczne, choć występuje pewna elastyczność konfiguracji [50, 51].

Baza Riak utrzymuje całą komunikację między klientami a węzłami z bazą jako lokalną w ramach jednego centrum danych. Dlatego  $n$  oznacza liczbę replik w ramach jednego centrum danych. Replikacja między klastrami bazy z różnych centrów danych odbywa się asynchronicznie w ten sposób podobny jak w replikacji z wieloma liderami [52].

## Wykrywanie jednoczesnych zapisów

Bazy w stylu Dynamo umożliwiają kilku klientom jednoczesny zapis wartości tego samego klucza. To oznacza, że konflikty pojawiają się nawet wtedy, gdy są używane kwora ściśle. To sytuacja podobna do tej w replikacji z wieloma liderami (zob. punkt „Radzenie sobie z konfliktami przy zapisie”), choć w bazach w stylu Dynamo konflikty mogą występować także w ramach uspoźniania przy odczycie lub w mechanizmie hinted-handoff.

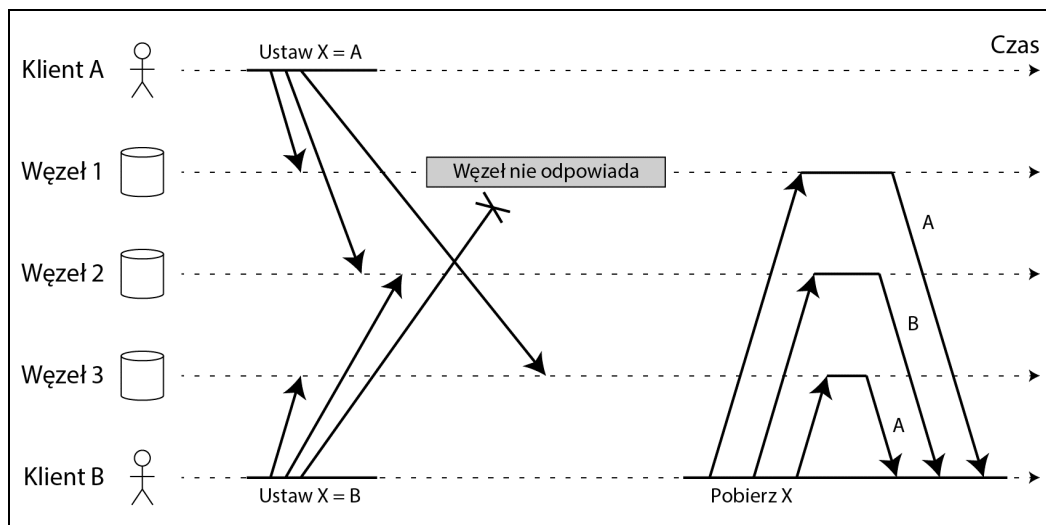
Problem polega na tym, że zdarzenia w różnych węzłach mogą zachodzić w różnej kolejności. Wynika to ze zmiennego opóźnienia w sieci i częściowych awarii. Na przykład na rysunku 5.12 są pokazane dwa klienty, A i B, jednocześnie zapisujące wartość klucza X w bazie danych z trzema węzłami:

- Węzeł 1 otrzymuje zapis od klienta A, ale z powodu tymczasowego przestoju nigdy nie otrzymuje zapisu od klienta B.
- Węzeł 2 najpierw otrzymuje zapis od klienta A, a następnie od klienta B.
- Węzeł 3 najpierw otrzymuje zapis od klienta B, a następnie od klienta A.

Gdyby każdy węzeł po prostu nadpisywał wartość klucza za każdym razem, gdy otrzyma od klienta żądanie zapisu, węzły byłyby stale niespójne, na co wskazuje końcowe żądanie *pobierz* z rysunku 5.12. Węzeł 2 przyjmuje, że końcową wartością klucza X jest B, natomiast pozostałe węzły uznają, że tą wartością jest A.

Aby uzyskać spójność ostateczną, repliki powinny dążyć do uzgodnienia tej samej wartości. Jak to robią? Można by mieć nadzieję, że bazy z replikacją radzą sobie z tym automatycznie. Jednak, niestety, większość implementacji jest niskiej jakości. Aby uniknąć utraty danych, programista aplikacji musi dobrze znać wewnętrzne mechanizmy obsługi konfliktów w używanej bazie.

Niektóre techniki rozwiązywania konfliktów pokrótce opisano w punkcie „Radzenie sobie z konfliktami przy zapisie”. Przed podsumowaniem tego rozdziału warto się bliżej przyjrzeć temu zagadnieniu.



Rysunek 5.12. Jednoczesne zapisy w bazie w stylu Dynamo. Nie występuje tu dobrze zdefiniowana kolejność

### Ostatni zapis wygrywa (odrzucając jednocześnie zapisy)

Jednym ze sposobów na uzyskanie spójności ostatecznej jest zadeklarowanie, że każda replika musi przechowywać tylko „najnowszą” wartość, oraz umożliwienie nadpisywania i odrzucania „starszych” wartości. Wtedy dopóki istnieje sposób na jednoznaczne ustalenie, który zapis jest „nowszy”, a każdy zapis jest ostatecznie kopiowany do wszystkich replik, repliki ostatecznie otrzymają tę samą wartość.

Cudzyśłów przy określeniu „najnowsza” wskazuje na to, że ta nazwa bywa myląca. W przykładzie z rysunku 5.12 żaden klient nie wiedział o drugim, gdy przysyłał żądanie zapisu do węzłów bazy. Nie jest więc jasne, który zapis był pierwszy. W rzeczywistości nie ma nawet sensu twierdzić, że któryś z zapisów nastąpił jako „pierwszy”. Zapisy są *jednoczesne*, dlatego ich kolejność jest nie-zdefiniowana.

Choć zapisy nie mają naturalnej kolejności, można wymusić ich arbitralne uporządkowanie. Możesz np. przypisać do każdego zapisu znacznik czasu, wybrać największy znacznik jako „najnowszy” i odrzucić wszystkie zapisy z wcześniejszymi znacznikami. Ten algorytm rozwiązywania konfliktów, nazywany *ostatni zapis wygrywa* (ang. *last write wins* — **LWW**), jest jedyną metodą usuwania konfliktów obsługiwaną w bazie Cassandra [53] i opcjonalną funkcją w bazie Riak [35].

Algorytm LWW zapewnia spójność ostateczną, jednak kosztem trwałości. Jeśli dla tego samego klucza wprowadzanych jest kilka jednoczesnych zapisów, to nawet jeżeli wszystkie zapisy zostały uznane za udane (ponieważ zapisano je w replikach), tylko jeden z nich przetrwa, a pozostałe zostaną odrzucone. Ponadto LWW może nawet usuwać zapisy, które nie są jednoczesne, co omówiono w punkcie „Znaczniki czasu do porządkowania zdarzeń”.

W niektórych sytuacjach, np. w kontekście pamięci podręcznej, utrata zapisów jest akceptowalna. Jeśli dzieje się inaczej, LWW nie nadaje się do rozwiązywania konfliktów.



Jedyny bezpieczny sposób używania bazy razem z algorytmem LWW to upewnienie się, że klucz jest zapisywany tylko raz, a następnie traktowany jak niemodyfikowalny. Pozwala to uniknąć jednoczesnych aktualizacji tego samego klucza. Na przykład zalecany sposób używania bazy Cassandra to stosowanie klucza w postaci identyfikatora UUID, co zapewnia każdej operacji unikatowy klucz [53].

### Relacja „zdarzyło się wcześniej” i równoległość

Jak zdecydować, czy dwie operacje są jednoczesne, czy nie? Aby to zrozumieć, przyjrzyj się przykładom:

- Dwa zapisy z rysunku 5.9 nie są jednoczesne. Wstawienie danych przez A *zdarzyło się wcześniej* niż zwiększenie wartości przez B, ponieważ wartość zwiększana przez B to wartość wstawiona przez A. Oznacza to, że operacja B to rozwinięcie operacji A. Tak więc operacja B musiała się zdarzyć później. Mówi się też, że operacja B jest *przyczynowo zależna* od A.
- Z kolei dwa zapisy z rysunku 5.12 są jednoczesne. Gdy każdy z klientów rozpoczyna operację, nie wie, że drugi klient też wykonuje działania na tym samym kluczu. Dlatego nie zachodzi zależność przyczynowa między operacjami.

Operacja A *zdarza się wcześniej* niż inna operacja B, jeśli B wie o A lub zależy od A, lub rozwija ją w jakiś sposób. To, czy dana operacja zdarza się wcześniej niż inna, jest istotą w definiowaniu znaczenia jednoczesności. Można po prostu stwierdzić, że dwie operacje są *jednoczesne* (współbieżne), jeśli żadna z nich nie zdarza się wcześniej niż druga (żadna nie wie o drugiej) [54].

Dlatego zawsze, gdy występują dwie operacje A i B, są trzy możliwości: albo A zdarzyła się przed B, albo B zdarzyła się przed A, albo A i B są jednoczesne. Potrzebny jest algorytm informujący, czy dwie operacje są jednoczesne, czy nie. Jeśli jedna operacja zdarzyła się wcześniej niż druga, późniejsza operacja powinna nadpisywać wcześniejszą. Jeżeli jednak operacje są jednoczesne, występuje konflikt, który trzeba rozwiązać.

## Jednoczesność, czas i względność

Może się wydawać, że dwie operacje należy nazywać jednoczesnymi, gdy zdarzają się „w tym samym czasie”. Jednak w rzeczywistości nie jest ważne, czy dosłownie się pokrywają. Z powodu problemów z zegarami w systemach rozproszonych trudno jest stwierdzić, czy dwie rzeczy dzieją się w tym samym czasie. To zagadnienie omówiono szczegółowo w rozdziale 8.

W ramach definiowania jednoczesności dokładny czas nie ma znaczenia. Dwie operacje są nazywane jednoczesnymi, jeśli nie wiedzą o sobie. Fizyczny czas ich wystąpienia nie jest istotny. Ludzie czasem łączą tę zasadę ze szczególną teorią względności w fizyce [54], w której wprowadzono ideę, że informacje nie mogą poruszać się szybciej niż z prędkością światła. Dlatego dwa zdarzenia zachodzące w określonej odległości od siebie nie mogą wywierać na siebie wpływu, jeśli czas między tymi zdarzeniami trwa krócej niż czas potrzebny na pokonanie odległości między nimi przez światło.

W systemach komputerowych dwie operacje mogą być jednoczesne nawet wtedy, gdy szybkość światła teoretycznie umożliwia wpływ jednej operacji na drugą. Na przykład jeśli sieć działa powoli lub występują w niej zakłócenia, dwie operacje mogą występować w różnym czasie i nadal być jednoczesne, ponieważ problemy w sieci sprawiają, że jedna operacja nie może się dowiedzieć o drugiej.

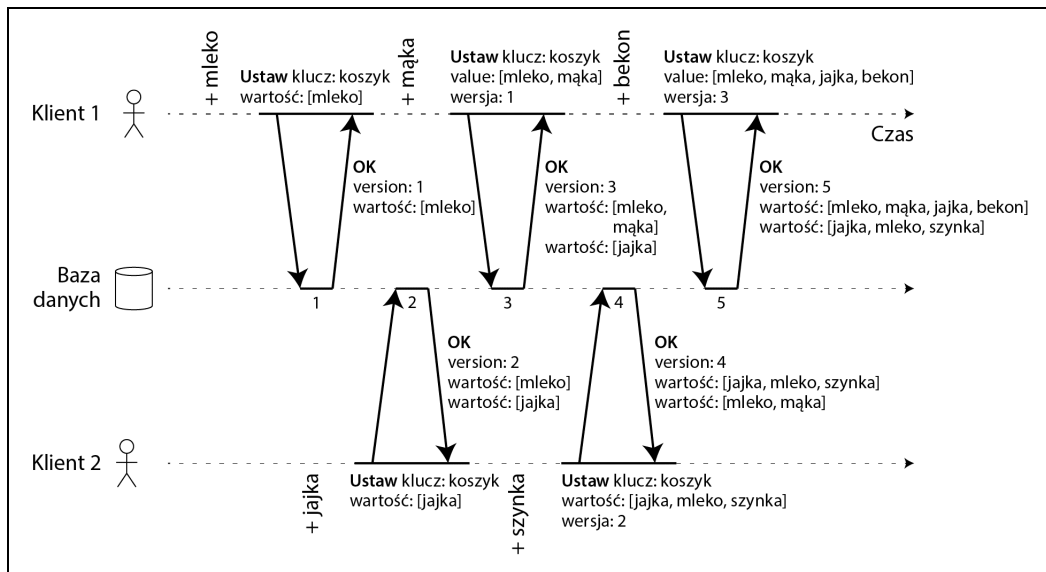
## Ujmowanie relacji „zdarzyło się wcześniej”

Przyjrzyj się algorytmowi, który określa, czy dwie operacje są jednoczesne lub czy jedna zdarzyła się przed drugą. Aby uprościć sytuację, zacznijmy od bazy z tylko jedną repliką. Po ustaleniu, jak wykonać zadanie z tylko jedną repliką, można uogólnić to podejście na bazy bez lidera z wieloma replikami.

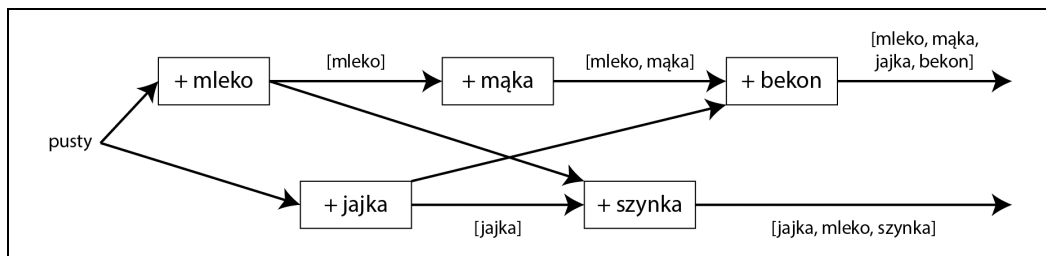
Na rysunku 5.13 pokazano dwa klienty jednocześnie dodające produkty do tego samego koszyka zakupów. (Jeśli ten przykład wydaje Ci się zbyt niedorzeczny, wyobraź sobie dwóch kontrolerów ruchu lotniczego jednocześnie dodających samolot do sektora, którym się opiekują). Początkowo koszyk jest pusty. Następnie klienty wykonują pięć zapisów w bazie:

1. Klient 1 dodaje do koszyka mleko. To pierwszy zapis z użyciem danego klucza, dlatego serwer z powodzeniem zapisuje dane i przypisuje im wersję 1. Serwer zwraca też klientowi wartość wraz z numerem wersji.
2. Klient 2 dodaje do koszyka jajka, nie wiedząc, że klient 1 jednocześnie dodał mleko (klient 2 przyjmuje, że jajka to jedyny produkt w koszyku). Serwer przypisuje wersję 2 do zapisanych danych i zachowuje jajka i mleko jako dwie odrębne wartości. Następnie zwraca klientowi *obie* te wartości wraz z numerem wersji 2.
3. Klient 1, nieświadom zapisu wprowadzonego przez klienta 2, chce dodać do koszyka mąkę. Dlatego uważa, że aktualną zawartością koszyka powinno być [mleko, mąka]. Przesła tę wartość na serwer wraz z numerem wersji 1, który to numer serwer wcześniej przekazał klientowi 1. Serwer na podstawie numeru wersji stwierdza, że zapis [mleko, mąka] zastępuje wcześniejszą wartość [mleko], jest jednak jednoczesny z wartością [jajka]. Dlatego serwer przypisuje wersję 3 do pary [mleko, mąka], zastępując wartość [mleko] (wersja 1), ale zachowując wartość [jajka] (wersja 3) i zwraca obie zachowane wartości klientowi.
4. W międzyczasie klient 2 chce dodać do koszyka szynkę. Nie wie jednak, że klient 1 właśnie dodał mąkę. Klient 2 w ostatniej odpowiedzi otrzymał od serwera dwie wartości, [mleko] i [jajka], dlatego teraz scala te wartości i dodaje szynkę, aby utworzyć nową wartość [jajka, mleko, szynka]. Przesła tę wartość na serwer wraz z wcześniejszym numerem wersji 2. Serwer wykrywa, że wersja 2 zastępuje wartość [jajka], jednak pojawia się jednocześnie z wartością [mleko, mąka]. Dlatego zachowywane wartości to [mleko, mąka] (wersja 3) i [jajka, mleko, szynka] (wersja 4).
5. W ostatnim kroku klient 1 chce dodać bekon. Wcześniej otrzymał z serwera wartości [mleko, mąka] i [jajka] (wersja 3), dlatego scala je, dodaje bekon i przesyła na serwer końcową wartość [mleko, mąka, jajka, bekon] wraz z numerem wersji 3. Nowe dane zastępują parę [mleko, mąka] (zauważ, że wartość [jajka] została już zastąpiona w poprzednim kroku), są jednak jednoczesne z wartością [jajka, mleko, szynka]. Dlatego serwer zachowuje te dwie jednoczesne wartości.

Przepływ danych między operacjami z rysunku 5.13 został graficznie przedstawiony na rysunku 5.14. Strzałki określają, która operacja *zdarzyła się wcześniej* względem innych operacji (w tym sensie, że późniejsza operacja *wiedziała o* wcześniejszej lub *zależała* od niej). W tym przykładzie klienty nigdy nie mają w pełni aktualnych danych z serwera, ponieważ zawsze jednocześnie wykonywana jest inna operacja. Jednak starsze wersje wartości ostatecznie są zastępowane i żaden zapis nie zostaje utracony.



Rysunek 5.13. Ujmowanie zależności przyczynowej między dwoma klientami jednocześnie edytującymi koszyk zakupów



Rysunek 5.14. Graf zależności przyczynowych z rysunku 5.13

Zauważ, że serwer może określić, czy dwie operacje są jednocześnie, sprawdzając numery wersji. Nie musi sam interpretować wartości (dlatego wartością może być dowolna struktura danych). Algorytm działa w następujący sposób:

- Serwer przechowuje numer wersji dla każdego klucza, zwiększa ten numer przy każdym zapisie wartości dla danego klucza i zachowuje nowy numer wersji wraz z zapisaną wartością.
- Gdy klient wczytuje klucz, serwer zwraca wszystkie wartości, które nie zostały zastąpione, a także najnowszy numer wersji. Klient musi wczytać klucz przed zapisaniem wartości.
- Gdy klient zapisuje wartość dla klucza, musi podać numer wersji z poprzedniego odczytu i scalić wszystkie wartości otrzymane we wcześniejszym odczycie. Odpowiedź na żądanie zapisu może wyglądać jak przy odczycie i obejmować wszystkie aktualne wartości. Pozwala to połączyć kilka zapisów w łańcuch — tak jak w przykładzie z koszykiem zakupów.
- Gdy serwer otrzymuje zapis z określonym numerem wersji, może zastąpić wszystkie wartości dla tego numeru i dla starszych wersji (ponieważ wie, że te dane zostały scalone w nową wartość). Musi jednak zachować wszystkie wartości o wyższym numerze wersji (ponieważ są one jednocześnie względem otrzymanego zapisu).

Gdy zapis obejmuje numer wersji z wcześniejszego odczytu, stanowi to informację, którego wcześniejszego stanu dotyczy dany zapis. Jeśli zapisujesz dane bez podawania numeru wersji, ta operacja jest jednoczesna względem wszystkich pozostałych zapisów, dlatego nie skutkuje zastąpieniem żadnych wartości. Dane są wtedy zwracane w dalszych odczytach jako jedna z wartości.

### Scalanie jednocześnie zapisanych wartości

Opisany algorytm gwarantuje, że żadne dane nie są niezauważalnie usuwane. Niestety wymaga jednak, by klienci wykonywały dodatkową pracę. Jeśli kilka operacji jest wykonywanych jednocześnie, klienci muszą później uporządkować je, scalając jednocześnie zapisane wartości. W bazie Riak jednocześnie zapisywane wartości są nazywane *siostrzanymi*.

Scalanie siostrzanych wartości to w istocie ten sam problem co opisane wcześniej rozwiązywanie konfliktów w replikacji z wieloma liderami (zob. punkt „Radzenie sobie z konfliktami przy zapisie”). Proste podejście polega na wyborze jednej z wartości na podstawie numeru wersji lub znacznika czasu (ostatni zapis wygrywa), co jednak może skutkować utratą danych. Dlatego w kodzie aplikacji konieczne może być zastosowanie bardziej inteligentnego rozwiązania.

W przykładzie koszyka zakupów sensownym sposobem scalania wartości siostrzanych jest wyznaczenie sumy zbiorów. Na rysunku 5.14 dwie końcowe wartości siostrzane to [mleko, mąka, jajka, bekon] i [jajka, mleko, szynka]. Zauważ, że mleko i jajka występują w obu zbiorach, choć zostały zapisane tylko jednokrotnie. Scalona wartość może mieć postać [mleko, mąka, jajka, bekon, szynka] (bez powtórzeń).

Jeśli jednak chcesz umożliwić użytkownikom także usuwanie produktów z koszyków (oprócz samego dodawania), wtedy wyznaczenie sumy zbiorów może nie wystarczyć. Jeżeli scalisz dwa siostrzane koszyki, a produkt zostanie usunięty tylko z jednego z nich, skasowany element pojawi się w sumie koszyków [37]. Aby zapobiec temu problemowi, produktu nie można w prosty sposób usuwać z bazy w momencie kasowania go. Zamiast tego system musi pozostawić znacznik z odpowiednim numerem wersji, aby określić, że produkt został usunięty. Taki znacznik usuwania nazywa się *nagrobkiem*. Wcześniej z nagrobkami zetknąłeś się w kontekście kompresji dzienników w punkcie „Indeksy z haszowaniem”.

Ponieważ scalanie wartości siostrzanych w kodzie aplikacji jest skomplikowane i narażone na błędy, podejmowane są wysiłki, aby projektować struktury danych, które potrafią automatycznie scalać dane. Opisano to w punkcie „Automatyczne rozwiązywanie konfliktów”. Na przykład system typów danych z bazy Riak umożliwia zastosowanie rodziny struktur danych CRDT [38, 39, 55], które potrafią automatycznie scalać wartości siostrzane w sensowny sposób (w tym z uwzględnieniem usuwanych danych).

### Wektory wersji

W przykładzie z rysunku 5.13 używa się tylko jednej repliki. W jaki sposób algorytm się zmienia, gdy występuje wiele replik, a nie ma lidera?

Na rysunku 5.13 do ujmowania zależności między operacjami stosuje się jeden numer wersji, co jednak nie wystarcza, gdy wiele replik jednocześnie akceptuje zapisy. Zamiast tego trzeba stosować numer wersji *dla repliki*, a także dla klucza. Każda replika zwiększa własny numer wersji, gdy

przetwarza zapis, a ponadto śledzi numery wersji zaobserwowane w innych replikach. Te informacje określają, które wartości należy zastąpić, a które zachować jako siostrzane.

Zestaw numerów wersji z wszystkich replik nazywa się *wektorem wersji* [56]. Stosuje się kilka wersji tego podejścia. Najciekawszy to prawdopodobnie *punktowy wektor wersji* (ang. *dotted version vector*) [57] stosowany w bazie Riak 2.0 [58, 59]. Szczegóły nie są tu omawiane, jednak taki wektor działa podobnie jak rozwiązanie w przykładowym koszyku.

Wektory wersji (podobnie jak numery wersji z rysunku 5.13) są przesyłane z replik bazy do klientów w momencie odczytu wartości i muszą być przesyłane z powrotem do bazy w czasie późniejszego zapisu danych. W bazie Riak wektor wersji jest kodowany jako łańcuch znaków nazywany *kontekstem przyczynowym*. Wektor wersji umożliwia bazie odróżnianie zapisów jednoczesnych od tych zastępujących dane.

Ponadto, też jak w przykładzie z jedną repliką, aplikacja musi czasem scalać wartości siostrzane. Struktura wektora wersji gwarantuje, że można bezpiecznie wczytać dane z jednej repliki, a następnie zapisać je w innej replice. Może to skutkować powstawaniem wartości siostrzanych, jeśli jednak są one poprawnie scalane, nie prowadzi to do utraty wartości.



#### Wektory wersji a zegary wektorowe

*Wektor wersji* jest czasem nazywany *zegarem wektorowym* (ang. *vector clock*), choć nie są one tym samym. Różnica między nimi jest subtelna. Szczegóły znajdziesz w literaturze cytowanej [57, 60, 61]. W skrócie można stwierdzić, że na potrzeby porównywania stanu replik odpowiednią strukturą danych są wektory wersji.

## Podsumowanie

W tym rozdziale opisana została replikacja. Może ona służyć kilku celom. Oto one:

### *Zapewnianie wysokiej dostępności*

Utrzymywanie działającego systemu nawet wtedy, gdy jedna maszyna (lub kilka, lub całe centrum danych) przestanie działać.

### *Praca bez połączenia z siecią*

Umożliwianie aplikacji kontynuowania pracy w sytuacji, gdy wystąpią zakłócenia w sieci.

### *Opóźnienia*

Rozmieszczanie danych blisko geograficznie do użytkowników. Użytkownicy mogą dzięki temu szybciej się komunikować z danymi.

### *Skalowalność*

Obsługiwanie — dzięki odczytom z replik — większej liczby odczytów niż za pomocą jednej maszyny.

Choć cel jest prosty (umieszczenie kopii tych samych danych na kilku maszynach), replikacja okazuje się zaskakująco skomplikowanym problemem. Wymaga starannego przemyślenia równoległości i wszystkich kwestii, które mogą się nie powieść, a także radzenia sobie z konsekwencjami

błędów. Trzeba uwzględnić przynajmniej niedostępne węzły i zakłócenia w sieci (i to nawet pomijając bardziej podstępne rodzaje błędów, takie jak ciche uszkodzenia danych z powodu błędów programowych).

Omówiono tu trzy główne podejścia do replikacji:

#### *Replikacja z jednym liderem*

Klienci przesyłają wszystkie zapisy do jednego węzła (lidera), który przekazuje strumień zmian danych do innych replik (obserwatorów). W odczytach można korzystać z dowolnych replik, przy czym odczyty z obserwatorów mogą być nieaktualne.

#### *Replikacja z wieloma liderami*

Klienci przesyłają każdy zapis do jednego z kilku węzłów-liderów. Każdy z liderów może akceptować zapisy. Liderzy przesyłają strumienie zmian danych między sobą i do węzłów-obserwatorów.

#### *Replikacja bez lidera*

Klienci przesyłają każdy zapis do kilku węzłów i wczytują dane równolegle z różnych węzłów, aby móc wykrywać i naprawiać węzły z nieaktualnymi danymi.

Każde z tych podejść ma wady i zalety. Replikacja z jednym liderem jest popularna, ponieważ można ją stosunkowo łatwo zrozumieć i nie trzeba się wtedy przejmować rozwiązywaniem konfliktów. Replikacja z wieloma liderami i bez lidera może zapewniać większą stabilność, gdy występują błędy węzłów, zakłócenia w sieci i skoki opóźnień. Koszt stanowi utrudnienie zrozumienia systemu i tylko bardzo słabe gwarancje spójności.

Replikacja może być synchroniczna i asynchroniczna. Ma to istotny wpływ na działanie systemu w momencie awarii. Choć replikacja asynchroniczna może być szybka, gdy system działa płynnie, ważne jest ustalenie, co się dzieje po wzroście opóźnień replikacji i awarii serwerów. Jeśli lider zawiedzie i promujesz na nowego lidera asynchronicznie aktualizowanego obserwatora, może to skutkować utratą niedawno zatwierdzonych danych.

Opisano tu dziwne efekty, które mogą być spowodowane opóźnieniem replikacji. Omówiono też kilka modeli zachowywania spójności przydatnych w trakcie decydowania, jak aplikacja powinna działać w sytuacji opóźnienia replikacji:

#### *Spójność odczytu po zapisie*

Użytkownicy zawsze powinni widzieć dane przesłane przez nich samych.

#### *Odczyty monotoniczne*

Gdy użytkownicy zobaczą dane z określonego momentu, nie powinni później widzieć danych z wcześniejszego czasu.

#### *Odczyty ze spójnym przedrostkiem*

Użytkownicy powinni widzieć dane, które mają sens przyczynowo (np. pytanie i odpowiedź na nie w odpowiedniej kolejności).

W końcowych punktach opisano kwestie związane z jednoczesnością nieodłączone od replikacji z wieloma liderami i bez lidera. Ponieważ te podejścia umożliwiają jednoczesny zapis wielu wartości, możliwe są konflikty. Omówiono tu algorytm, który można zastosować w bazach do określania, czy jedna operacja zdarzyła się wcześniej od innej lub czy nastąpiły one jednocześnie. Poruszono też temat metod rozwiązywania konfliktów przez scalanie jednoczesnych aktualizacji.

W następnym rozdziale dalej opisywane są dane rozproszone między wiele maszyn z użyciem podejścia innego od replikacji — podziału dużych zbiorów danych na *partycje*.

## **Literatura cytowana**

- [1] Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri i in., *Notes on Distributed Databases*, „IBM Research”, Research Report RJ2571 (33471), lipiec 1979 ([http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/\\$File/RJ2571.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf)).
- [2] Oracle Active Data Guard Real-Time Data Protection and Availability, „Oracle White Paper”, czerwiec 2013 (<http://www.oracle.com/technetwork/database/availability/active-data-guard-wp-12c-1896127.pdf>).
- [3] AlwaysOn Availability Groups, „SQL Server Books Online”, Microsoft, 2012 (<https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/always-on-availability-groups-sql-server>).
- [4] Lin Qiao, Kapil Surlaker, Shirshanka Das i in., *On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2013 (<https://www.slideshare.net/amywtang/espresso-20952131>).
- [5] Jun Rao, *Intra-Cluster Replication for Apache Kafka*, w: „ApacheCon North America”, luty 2013 (<https://www.slideshare.net/junrao/kafka-replication-apachecon2013>).
- [6] *Highly Available Queues*, „RabbitMQ Server Documentation”, Pivotal Software, Inc., 2014 (<https://www.rabbitmq.com/ha.html>).
- [7] Yoshinori Matsunobu, *Semi-Synchronous Replication at Facebook*, yoshinorimatsunobu.blogspot.co.uk, 1 kwietnia 2014 (<http://yoshinorimatsunobu.blogspot.co.uk/2014/04/semi-synchronous-replication-at-facebook.html>).
- [8] Robbert van Renesse i Fred B. Schneider, *Chain Replication for Supporting High Throughput and Availability*, w: „6th USENIX Symposium on Operating System Design and Implementation” (OSDI), grudzień 2004 ([http://static.usenix.org/legacy/events/osdi04/tech/full\\_papers/renesse/renesse.pdf](http://static.usenix.org/legacy/events/osdi04/tech/full_papers/renesse/renesse.pdf)).
- [9] Jeff Terrace i Michael J. Freedman, *Object Storage on CRAQ: HighThroughput Chain Replication for Read-Mostly Workloads*, w: „USENIX Annual Technical Conference” (ATC), czerwiec 2009 ([https://www.usenix.org/legacy/event/usenix09/tech/full\\_papers/terrace/terrace.pdf](https://www.usenix.org/legacy/event/usenix09/tech/full_papers/terrace/terrace.pdf)).
- [10] Brad Calder, Ju Wang, Aaron Ogus i in., *Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency*, w: „23rd ACM Symposium on Operating Systems Principles” (SOSP), październik 2011 (<http://sigops.org/sosp/sosp11/current/2011-Cascais/printable/11-calder.pdf>).

- [11] Andrew Wang, *Windows Azure Storage*, umbrant.com, 4 lutego 2016 ([http://umbrant.com/blog/2016/windows\\_azure\\_storage.html](http://umbrant.com/blog/2016/windows_azure_storage.html)).
- [12] Percona Xtrabackup — Documentation, Percona LLC, 2014 (<https://www.percona.com/doc/percona-xtrabackup/2.1/index.html>).
- [13] Jesse Newland, *GitHub Availability This Week*, github.com, 14 września 2012 (<https://github.com/blog/1261-github-availability-this-week>).
- [14] Mark Imbriaco, *Downtime Last Saturday*, github.com, 26 grudnia 2012 (<https://github.com/blog/1364-downtime-last-saturday>).
- [15] John Hugg, „All in” with Determinism for Performance and Testing in Distributed Systems, w: „Strange Loop”, wrzesień 2015 (<https://www.youtube.com/watch?v=gJRj3vJL4wE>).
- [16] Amit Kapila, *WAL Internals of PostgreSQL*, w: „PostgreSQL Conference” (PGCon), maj 2012 ([http://www.pgcon.org/2012/schedule/attachments/258\\_212\\_Internals%20Of%20PostgreSQL%20Wal.pdf](http://www.pgcon.org/2012/schedule/attachments/258_212_Internals%20Of%20PostgreSQL%20Wal.pdf)).
- [17] *MySQL Internals Manual*, Oracle, 2014 (<https://dev.mysql.com/doc/internals/en/>).
- [18] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang i in., *Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services*, w: „12th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), maj 2015 (<https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf>).
- [19] *Oracle GoldenGate 12c: Real-Time Access to Real-Time Information*, Oracle White Paper, październik 2013 (<http://www.oracle.com/us/products/middleware/data-integration/oracle-goldengate-realtime-access-2031152.pdf>).
- [20] Shirshanka Das, Chavdar Botev, Kapil Surlaker i in., *All Aboard the Databus!*, w: „ACM Symposium on Cloud Computing” (SoCC), październik 2012 ([https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/s18-das.pdf?attachauth=ANoY7co5lojtLXkT3\\_lnGwp837-LPGkg21n07CGEkjlbImJteZDrclslA8mRUxuu84Vc6mXCB8mRE9y0wa-US0emrUREuSNkTp4MCDgzLJOztW0xfbHZzmjJWNC\\_Ph\\_FxHzZ57dT-QvkoySs4Tkxh2Q1V2pebvGTbWQOnpjgnimjkXO0x5EgJBM1XxlF9\\_ZpwDPfSlpm5fm1MGPixOmmmrJCau6hNzQrqrw%3D%3D&attredirects=0](https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/s18-das.pdf?attachauth=ANoY7co5lojtLXkT3_lnGwp837-LPGkg21n07CGEkjlbImJteZDrclslA8mRUxuu84Vc6mXCB8mRE9y0wa-US0emrUREuSNkTp4MCDgzLJOztW0xfbHZzmjJWNC_Ph_FxHzZ57dT-QvkoySs4Tkxh2Q1V2pebvGTbWQOnpjgnimjkXO0x5EgJBM1XxlF9_ZpwDPfSlpm5fm1MGPixOmmmrJCau6hNzQrqrw%3D%3D&attredirects=0)).
- [21] Greg Sabino Mullane, *Version 5 of Bucardo Database Replication System*, blog.endpoint.com, 23 czerwca 2014 (<https://www.endpoint.com/blog/2014/06/23/bucardo-5-multimaster-postgres-released>).
- [22] Werner Vogels, *Eventually Consistent*, „ACM Queue”, rocznik 6, nr 6, s. 14 – 19, październik 2008 (<http://queue.acm.org/detail.cfm?id=1466448>; <https://dl.acm.org/citation.cfm?doid=1466443.1466448>).
- [23] Douglas B. Terry, *Replicated Data Consistency Explained Through Baseball*, Microsoft Research, Technical Report MSR-TR-2011-137, październik 2011 (<https://www.microsoft.com/en-us/research/publication/replicated-data-consistency-explained-through-baseball/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F157411%2Fconsistencyandbaseballreport.pdf>).



- [24] Douglas B. Terry, Alan J. Demers, Karin Petersen i in., *Session Guarantees for Weakly Consistent Replicated Data*, w: „3rd International Conference on Parallel and Distributed Information Systems” (PDIS), wrzesień 1994 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.2269&rep=rep1&type=pdf>; <http://ieeexplore.ieee.org/document/331722/>).
- [25] Terry Pratchett, *Reaper Man: A Discworld Novel*, Victor Gollancz, 1991, ISBN: 978-0-575-04979-6.
- [26] *Tungsten Replicator*, Continuent, Inc., 2014 (<https://github.com/continuent>).
- [27] *BDR 0.10.0 Documentation*, The PostgreSQL Global Development Group, *bdr-project.org*, 2015 (<http://bdr-project.org/docs/next/index.html>).
- [28] Robert Hodges, *If You \*Must\* Deploy Multi-Master Replication, Read This First*, *scale-out-blog.blogspot.co.uk*, 30 marca 2012 (<http://scale-out-blog.blogspot.co.uk/2012/04/if-you-must-deploy-multi-master.html>).
- [29] J. Chris Anderson, Jan Lehnardt i Noah Slater, *CouchDB: The Definitive Guide*, O'Reilly Media, 2010, ISBN: 978-0-596-15589-6.
- [30] AppJet, Inc., *Etherpad and EasySync Technical Manual*, *github.com*, 26 marca 2011 (<https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf>).
- [31] John Day-Richter, *What's Different About the New Google Docs: Making Collaboration Fast*, *googledrive.blogspot.com*, 23 września 2010 (<https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html>).
- [32] Martin Kleppmann i Alastair R. Beresford, *A Conflict-Free Replicated JSON Datatype*, arXiv:1608.03960, 13 sierpnia 2016 (<https://arxiv.org/abs/1608.03960>).
- [33] Frazer Clement, *Eventual Consistency — Detecting Conflicts*, *messagepassing.blogspot.co.uk*, 20 października 2011 (<http://messagepassing.blogspot.co.uk/2011/10/eventual-consistency-detecting.html>).
- [34] Robert Hodges, *State of the Art for MySQL Multi-Master Replication*, w: „Percona Live: MySQL Conference & Expo”, kwiecień 2013 (<https://www.percona.com/live/mysql-conference-2013/sessions/state-art-mysql-multi-master-replication>).
- [35] John Daily, *Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems*, *basho.com*, 12 listopada 2013 (<http://basho.com/posts/technical/clocks-are-bad-or-welcome-to-distributed-systems/>).
- [36] Riley Berton, *Is Bi-Directional Replication (BDR) in Postgres Transactional?*, *sdf.org*, 4 stycznia 2016 (<http://sdf.org/~riley/blog/2016/01/04/is-bi-directional-replication-bdr-in-postgres-transactional/>).
- [37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani i in., *Dynamo: Amazon's Highly Available Key-Value Store*, w: „21st ACM Symposium on Operating Systems Principles” (SOSP), październik 2007 (<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>).

- [38] Marc Shapiro, Nuno Preguiça, Carlos Baquero i Marek Zawirski, *A Comprehensive Study of Convergent and Commutative Replicated Data Types*, INRIA Research Report nr 7506, styczeń 2011 (<https://hal.inria.fr/inria-00555588/>).
- [39] Sam Elliott, *CRDTs: An UPDATE (or Maybe Just a PUT)*, w: *RICON West*, październik 2013 (<https://speakerdeck.com/lenary/crdts-an-update-or-just-a-put>).
- [40] Russell Brown, *A Buffers Guide to CRDTs in Riak*, [gist.github.com](https://gist.github.com/russelldb/f92f44bdfb619e089a4d), 28 października 2013 (<https://gist.github.com/russelldb/f92f44bdfb619e089a4d>).
- [41] Benjamin Farinier, Thomas Gazagnaire i Anil Madhavapeddy, *Mergeable Persistent Data Structures*, w: „26es Journées Francophones des Langages Applicatifs” (JFLA), styczeń 2015 (<http://gazagnaire.org/pub/FGM15.pdf>).
- [42] Chengzheng Sun i Clarence Ellis, *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements*, w: „ACM Conference on Computer Supported Cooperative Work” (CSCW), listopad 1998 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.933&rep=rep1&type=pdf>).
- [43] Lars Hofhansl, *HBASE-7709: Infinite Loop Possible in Master/Master Replication*, [issues.apache.org](https://issues.apache.org/jira/browse/HBASE-7709), 29 stycznia 2013 (<https://issues.apache.org/jira/browse/HBASE-7709>).
- [44] David K. Gifford, *Weighted Voting for Replicated Data*, w: „7th ACM Symposium on Operating Systems Principles” (SOSP), grudzień 1979 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.7698>; <https://dl.acm.org/citation.cfm?doid=800215.806583>).
- [45] Heidi Howard, Dahlia Malkhi i Alexander Spiegelman, *Flexible Paxos: Quorum Intersection Revisited*, [arXiv:1608.06696](https://arxiv.org/abs/1608.06696), 24 sierpnia 2016 (<https://arxiv.org/abs/1608.06696>).
- [46] Joseph Blomstedt, *Re: Absolute Consistency*, e-mail na liście dyskusyjnej *riak-users*, [lists.basho.com](https://lists.basho.com/pipermail/riak-users_lists.basho.com/2012-January/007157.html), 11 stycznia 2012 ([http://lists.basho.com/pipermail/riak-users\\_lists.basho.com/2012-January/007157.html](http://lists.basho.com/pipermail/riak-users_lists.basho.com/2012-January/007157.html)).
- [47] Joseph Blomstedt, *Bringing Consistency to Riak*, w: *RICON West*, październik 2012 (<https://vimeo.com/51973001>).
- [48] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin i in., *Quantifying Eventual Consistency with PBS*, „Communications of the ACM”, rocznik 57, nr 8, s. 93 – 102, sierpień 2014 (<http://www.bailis.org/papers/pbs-cacm2014.pdf>; <https://dl.acm.org/citation.cfm?doid=2632661.2632792>).
- [49] Jonathan Ellis, *Modern Hinted Handoff*, [datastax.com](https://www.datastax.com/dev/blog/modern-hinted-handoff), 11 grudnia 2012 (<https://www.datastax.com/dev/blog/modern-hinted-handoff>).
- [50] *Project Voldemort Wiki*, [github.com](https://github.com/voldemort/voldemort/wiki), 2013 (<https://github.com/voldemort/voldemort/wiki>).
- [51] *Apache Cassandra 2.0 Documentation*, DataStax, Inc., 2014 (<http://docs.datastax.com/en/archived/cassandra/2.0/index.html>).
- [52] *Riak Enterprise: Multi-Datacenter Replication*, tekst techniczny, Basho Technologies, Inc., wrzesień 2014.

- [53] Jonathan Ellis, *Why Cassandra Doesn't Need Vector Clocks*, *datastax.com*, 2 września 2013 (<https://www.datastax.com/dev/blog/why-cassandra-doesnt-need-vector-clocks>).
- [54] Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, „Communications of the ACM”, rocznik 21, nr 7, s. 558 – 565, lipiec 1978 (<https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Ftime-clocks.pdf>; <https://dl.acm.org/citation.cfm?doid=359545.359563>).
- [55] Joel Jacobson, *Riak 2.0: Data Types*, *blog.joeljacobsen.com*, 23 marca 2014.
- [56] D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin i in., *Detection of Mutual Inconsistency in Distributed Systems*, „IEEE Transactions on Software Engineering”, rocznik 9, nr 3, s. 240 – 247, maj 1983 (<http://zoo.cs.yale.edu/classes/cs426/2013/bib/parker83detection.pdf>).
- [57] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida i in., *Dotted Version Vectors: Logical Clocks for Optimistic Replication*, *arXiv:1011.5808*, 26 listopada 2010 (<https://arxiv.org/pdf/1011.5808v1.pdf>).
- [58] Sean Cribbs, *A Brief History of Time in Riak*, w: *RICON*, październik 2014.
- [59] Russell Brown, *Vector Clocks Revisited Part 2: Dotted Version Vectors*, *basho.com*, 10 listopada 2015 (<http://basho.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/>).
- [60] Carlos Baquero, *Version Vectors Are Not Vector Clocks*, *haslab.wordpress.com*, 8 lipca 2011 (<https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/>).
- [61] Reinhard Schwarz i Friedemann Mattern, *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail*, „Distributed Computing”, rocznik 7, nr 3, s. 149 – 174, marzec 1994 (<https://disco.ethz.ch/courses/hs08/seminar/papers/mattern4.pdf>; <https://link.springer.com/article/10.1007%2FBF02277859>).

Elasticsearch

PODZIAŁ NA  
PODSTAWIE  
DOKUMENTÓW

Solr

INDEKSY POMOCNICZE

PODZIAŁ NA  
PODSTAWIE POJĘĆ

Voldemort

PODZIAŁ WEDŁUG SKRÓTÓW

Cassandra

MongoDB

HBase

ATOL PODZIAŁU NA PARTYCJE

PODZIAŁ  
ZAKRESÓW KLUCZY

RethinkDB

ROWNOWAŻENIE

TRASOWANIE ZĄDANY

# Podział na partycje

*Najwyraźniej musimy odejść od sekwencyjności i nie ograniczać komputerów. Musimy określić definicje i ustalić priorytety oraz opisy danych. Musimy określić relacje, a nie procedury.*

— Grace Murray Hopper, *Management and the Computer of the Future* (1962)

W rozdziale 5. omówiono replikację — tworzenie wielu kopii tych samych danych w różnych węzłach. Dla bardzo dużych zbiorów danych lub przy bardzo wysokiej liczbie zapytań to nie wystarczy. Trzeba podzielić dane na *partycje* (ta technika jest też nazywana podziałem — ang. *sharding*)<sup>1</sup>.



## Problemy z terminologią

Coś, co tu jest nazywane *partycją*, nosi nazwę *shard* w bazach MongoDB, Elasticsearch i SolrCloud, *region* w bazie HBase, *tablet* w bazie Bigtable, *vnode* w bazach Cassandra i Riak oraz *vBucket* w bazie Couchbase. Jednak *partycja* to najpowszechniej przyjęta nazwa, dlatego będziemy się jej trzymać.

Standardowo partycje są definiowane w taki sposób, aby każda porcja danych (każdy rekord, wiersz lub dokument) znajdowała się tylko w jednej partycji. Istnieją różne sposoby na uzyskanie tego efektu. Zostały one szczegółowo omówione w tym rozdziale. W efekcie każda partycja to mała samodzielna baza, przy czym cała baza może obsługiwać operacje używające jednocześnie wielu partycji.

Główną motywacją do podziału danych na partycje jest *skalowalność*. Różne partycje można umieścić w różnych węzłach w klastrze bez zasobów współdzielonych (definicję systemów *bez zasobów współdzielonych* znajdziesz we wprowadzeniu do części II). Dlatego duży zbiór danych można rozdzielić między wiele dysków, a obciążenie związane z zapytaniami — między wiele procesorów.

Zapytania operujące na jednej partycji każdy węzeł może wykonywać niezależnie z użyciem własnej partycji. Tak więc przepustowość obsługi zapytań można zwiększyć, dodając węzły. Duże, złożone zapytania można potencjalnie przetwarzać równolegle w wielu węzłach, choć jest to znacznie trudniejsze.

---

<sup>1</sup> Podział na partycje omawiany w tym rozdziale to sposób celowego podziału dużej bazy na mniejsze. Nie ma to nic wspólnego z *netsplitem* (podziałem sieci), czyli rodzajem błędu w sieci łączącej węzły. Takie błędy są opisane w rozdziale 8.

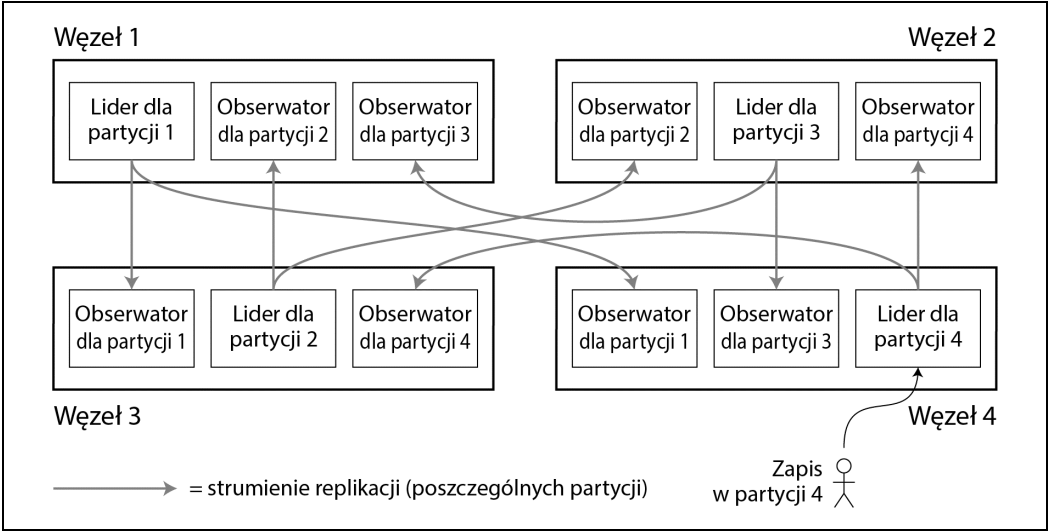
Bazy z podziałem na partycje po raz pierwszy wprowadzono w latach 80. w formie produktów takich jak Teradata i Tandem NonStop SQL [1]. W ostatnim czasie nastąpiło ich „ponowne odkrycie” w postaci baz typu NoSQL i hurtowni danych opartych na Hadoopie. Niektóre systemy są projektowane pod kątem transakcyjnego obciążenia roboczego, a inne z myślą o analityce (zob. punkt „Przetwarzanie transakcji czy analityka?”). Ta różnica wpływa na to, jak system jest dostrojony, jednak podstawy podziału na partycje dotyczą obu rodzajów obciążenia roboczego.

W tym rozdziale najpierw opisane są różne sposoby podziału dużych zbiorów danych na partycje. Zobaczysz też interakcje indeksowania danych z podziałem na partycje. Dalej omówiono równoważenie potrzebne w sytuacji, gdy chcesz dodać lub usunąć węzły w klastrze. Na zakończenie przedstawiono przegląd tego, jak bazy kierują zadaniami do odpowiednich partycji i wykonują zapytania.

## Podział na partycje i replikacja

Podział na partycje jest zwykle łączony z replikacją. Dlatego kopie każdej partycji są przechowywane w wielu węzłach. To oznacza, że choć każdy rekord znajduje się tylko w jednej partycji, może być przechowywany w kilku różnych węzłach w celu zapewnienia odporności na błędy.

Węzeł może przechowywać więcej niż jedną partycję. Jeśli używa się modelu replikacji lider-obszernik, połączenie podziału na partycje i replikacji może wyglądać tak jak na rysunku 6.1. Liderem każdej partycji jest jeden węzeł, a pozostałe węzły są obserwatorem. Każdy węzeł może być liderem dla niektórych partycji i obserwatorem dla pozostałych.



Rysunek 6.1. Łączenie replikacji i podziału na partycje. Każdy węzeł jest liderem dla niektórych partycji i obserwatorem dla innych

Wszystko, co napisano w rozdziale 5. na temat replikacji baz, w równym stopniu dotyczy także replikacji partycji. Wybór sposobu podziału na partycje jest w dużej mierze niezależny od używanego modelu replikacji. Dlatego aby uprościć omówienie, w tym rozdziale replikacja jest ignorowana.

# Podział na partycje danych typu klucz-wartość

Załóżmy, że masz dużą ilość danych i chcesz je podzielić. Jak zdecydować, które rekordy mają się znaleźć w poszczególnych węzłach?

Celem w trakcie podziału na partycje jest równomierne rozdzielenie między węzły danych i związanego z zapytaniami obciążenia. Jeśli każdy węzeł obsługuje taką samą część danych, wtedy (teoretycznie) 10 węzłów powinno móc obsłużyć 10 razy więcej danych i 10-krotnie większą przepustowość odczytów i zapisów niż jeden węzeł (na razie replikacja jest ignorowana).

Jeżeli podział nie jest równy i niektóre partycje obejmują więcej danych lub obsługują więcej zapytań niż inne, nazywamy go *asymetrycznym*. Występowanie asymetrii znacznie zmniejsza efektywność podziału. W skrajnych przypadkach całe obciążenie może być obsługiwane przez jedną partycję, dlatego 9 z 10 węzłów jest bezczynnych, a wąskie gardło stanowi jeden zajęty węzeł. Partycję z nieproporcjonalnie wysokim obciążeniem można nazwać *partycją przeciążoną* (ang. *hot spot*).

Najprostszy sposób unikania partycji przeciążonych to losowy przydział rekordów do węzłów. Wtedy dane są rozdzielone stosunkowo równomiernie między węzły. Ta technika ma jednak poważną wadę — gdy próbujesz wczytać konkretny element, nie wiesz, w którym węzle się znajduje. Dlatego musisz kierować zapytania równoległe do wszystkich węzłów.

Możliwe jest lepsze rozwiązanie. Załóżmy, że używa się prostego modelu danych klucz-wartość, w którym dostęp do rekordów zawsze odbywa się na podstawie klucza głównego. Na przykład w starożytnej drukowanej encyklopedii szukasz wpisów na podstawie tytułów. Ponieważ wszystkie wpisy są alfabetycznie posortowane według tytułu, możesz szybko znaleźć potrzebne Ci informacje.

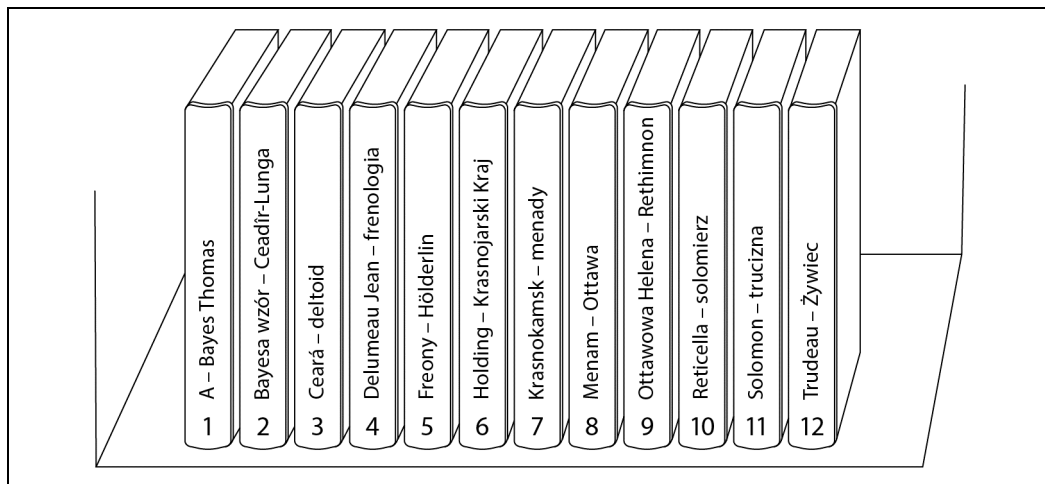
## Podział na podstawie przedziałów kluczy

Jeden ze sposobów podziału to przypisanie do każdej partycji ciągłego przedziału kluczy (od określonego minimum do maksimum) tak jak w tomach drukowanej encyklopedii (rysunek 6.2). Jeśli znasz granice przedziałów, możesz łatwo ustalić, która partycja zawiera dany klucz. Jeżeli wiesz też, które partycje są przypisane do poszczególnych węzłów, możesz kierować żądania bezpośrednio do odpowiedniego węzła (lub, w przypadku biblioteki, wybrać z półki właściwą książkę).

Przedziały kluczy nie muszą być sobie równe, ponieważ dane mogą nie być równomiernie rozdzielone. Na przykład na rysunku 6.2 tom 1 obejmuje słowa zaczynające się literami A i B, natomiast tom 12 zawiera wyrazy rozpoczynające się od liter T, U, V, X, Y i Z. Proste przypisanie do każdego tomu dwóch liter alfabetu spowodowałoby, że niektóre tomy byłyby znacznie obszerniejsze od pozostałych. Aby równomiernie rozdzielić dane, granice partycji trzeba dostosować do danych.

Granice partycji mogą być albo ręcznie ustalane przez administratora, albo dobierane automatycznie przez bazę (określanie granic partycji szczegółowo opisano w punkcie „Równoważenie partycji”). Ta strategia podziału na partycje jest używana w systemie Bigtable, jego otwartym odpowiedniku HBase [2, 3], bazie RethinkDB i w bazie MongoDB w wersjach starszych niż 2.4 [4].

W każdej partycji można przechowywać klucze zgodnie z porządkiem sortowania (zob. punkt „Pliki SSTable i drzewa LSM”). Ma to tę zaletę, że skanowanie przedziałów jest łatwe i można traktować klucz jak indeks połączony, aby pobrać kilka powiązanych rekordów w jednym zapytaniu



Rysunek 6.2. Drukowana encyklopedia jest podzielona według przedziałów kluczy

(zob. punkt „Indeksy wielokolumnowe”). Pomyśl np. o aplikacji, która przechowuje dane z sieci czujników, gdzie klucz to znacznik czasu pomiaru (*rok-miesiąc-dzień-godzina-minuta-sekunda*). W takiej sytuacji skanowanie przedziałów jest bardzo przydatne, ponieważ pozwala łatwo pobrać np. wszystkie odczyty z danego miesiąca.

Wadą podziału na podstawie przedziałów kluczy jest jednak to, że niektóre wzorce dostępu prowadzą do powstawania partycji przeciążonych. Jeśli klucz to znacznik czasu, partycje odpowiadają przedziałom czasu (np. jedna partycja dla każdego dnia). Niestety, ponieważ dane z czujników są zapisywane w bazie w momencie dokonywania pomiarów, wszystkie zapisy trafiają do tej samej partycji (przeznaczonej dla dzisiejszego dnia). Dlatego ta partycja jest przeciążona zapisami, natomiast pozostałe są bezczynne.

Aby uniknąć tego problemu w bazie ze wskazaniem czujników, pierwszym elementem klucza musi być coś innego niż znacznik czasu. Możesz np. poprzedzić każdy znacznik czasu nazwą czujnika, tak by podział odbywał się najpierw według tych nazw, a następnie według czasu. Przy założeniu, że w danym momencie aktywnych jest wiele czujników, obciążenie zapisami zostanie rozłożone bardziej równomiernie między partycje. Jednak gdy zechcesz pobrać wartości z wielu czujników z danego przedziału czasu, będziesz musiał wykonać odrębne zapytanie o przedział dla nazwy każdego czujnika.

## Podział na podstawie skrótów kluczy

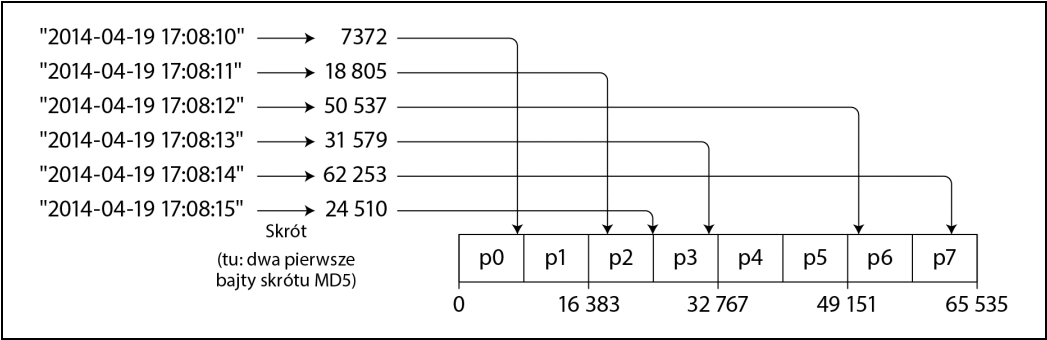
Z powodu zagrożenia asymetrią i hot spotami w wielu bazach rozproszonych do określania partycji dla danego klucza używa się funkcji skrótu.

Dobra funkcja skrótu przyjmuje asymetryczne dane i zapewnia ich równomierny podział. Załóżmy, że używasz 32-bitowej funkcji skrótu przyjmującej łańcuch znaków. Za każdym razem, gdy przekażesz jej nowy łańcuch, zwróci ona pozornie losową liczbę z przedziału od 0 do  $2^{32} - 1$ . Nawet jeśli wejściowe łańcuchy znaków są bardzo podobne, ich skróty są równomiernie rozdzielane w podanym przedziale wartości.



W kontekście podziału na partycje funkcja skrótu nie musi być kryptograficznie silna. Na przykład w bazach Cassandra i MongoDB używana jest funkcja MD5, a w bazie Voldemort — funkcja Fowlera-Nolla-Vo. Wiele języków programowania obejmuje wbudowane proste funkcje skrótu (ponieważ są one używane w tablicach z haszowaniem), które jednak mogą być nieodpowiednie do podziału na partycje. Na przykład funkcje `Object.hashCode()` z Javy i `Object#hash` z języka Ruby mogą zwracać inne skróty dla tych samych kluczy w różnych procesach [6].

Gdy masz już odpowiednią funkcję skrótu dla kluczy, możesz powiązać każdą partycję z przedziałem skrótów (zamiast z przedziałem kluczy), a każdy klucz, którego skrót znajduje się w przedziale dla danej partycji, zostanie w niej zapisany. Przedstawia to rysunek 6.3.



Rysunek 6.3. Podział na partycje według skrótu klucza

Ta technika dobrze się nadaje do równomiernego podziału kluczy między partycje. Granice partycji mogą tworzyć równe przedziały lub być dobierane pseudolosowo (wtedy ta technika jest czasem nazywana *spójnym haszowaniem*).

## Spójne haszowanie

Spójne haszowanie zdefiniowane przez Kargera i innych [7] to sposób na równomierny podział obciążenia w internetowym systemie pamięci podręcznej, np. w sieci CDN (ang. *content delivery network*). W tym podejściu używane są losowo dobrane granice partycji, aby uniknąć konieczności scentralizowanej kontroli lub osiągnięcia konsensusu w środowisku rozproszonym. Zauważ, że określenie *spójne* nie ma tu nic wspólnego ze spójnością replik (zob. rozdział 5.) lub spójnością w modelu ACID (zob. rozdział 7.). Określa tylko konkretny sposób równoważenia partycji.

W punkcie „Równoważenie partycji” zobaczysz, że to konkretne podejście nie jest wysoce skuteczne dla baz danych [8], dlatego w praktyce rzadko się je stosuje. W dokumentacji niektórych baz znajdują się informacje o spójnym haszowaniu, jednak często są one nieprecyzyjne. Ponieważ to mylące, najlepiej jest unikać pojęcia *spójne haszowanie* i stosować określenie *podział na partycje według skrótów*.

Niestety, podział na partycje na podstawie skrótów kluczy skutkuje utratą przydatnej cechy podziału według przedziałów kluczy — nie można wydajnie przetwarzać zapytań o przedziały. Klucze, które wcześniej były przyległe, są teraz rozproszone między wszystkie partycje, dlatego porządek sortowania zostaje utracony. W bazie MongoDB w trybie podziału na podstawie skrótów każde zapytanie o przedział musi być kierowane do wszystkich partycji [4]. Zapytania o przedziały oparte na kluczu głównym nie są obsługiwane w bazach Riak [9], Couchbase [10] ani Voldemort.

W bazie Cassandra osiągnięto kompromis między opisanymi strategiami [11, 12, 13]. W tej bazie można zadeklarować tabelę ze *złożonym kluczem głównym* składającym się z kilku kolumn. Haszowana jest tylko pierwsza część klucza w celu określenia partycji. Jednak pozostałe kolumny są używane jako indeks połączony na potrzeby sortowania danych w plikach SSTable tej bazy. Zapytanie nie może więc przeszukiwać przedziałów wartości z pierwszej kolumny klucza złożonego, jeśli jednak używa się stałej wartości pierwszej kolumny, możliwe jest wydajne skanowanie przedziałów według innych kolumn klucza.

Podejście z indeksem scalanym umożliwia stosowanie eleganckiego modelu danych dla relacji jeden do wielu. Na przykład w serwisie społecznościowym jeden użytkownik może zamieszczać wiele aktualizacji. Jeśli klucz główny aktualizacji to (user\_id, update\_timestamp), można wydajnie pobrać wszystkie aktualizacje wprowadzone przez danego użytkownika w określonym terminie posortowane według znacznika czasu. Dane różnych użytkowników mogą być przechowywane w innych partycjach, jednak aktualizacje konkretnego użytkownika są uporządkowane według znacznika czasu w jednej partycji.

## Asymetryczne obciążenie robocze i odciażanie hot spotów

Wcześniej opisano, że haszowanie klucza w celu wyznaczenia jego partycji może pomóc ograniczyć powstawanie hot spotów. Nie da się jednak całkowicie ich uniknąć. W skrajnych przypadkach, gdy wszystkie odczyty i zapisy dotyczą tego samego klucza, nadal wszystkie żądania są kierowane do tej samej partycji.

Tego rodzaju obciążenie robocze jest nietypowe, jednak może się zdarzyć. Na przykład w serwisie społecznościowym działania celebryty z milionami obserwatorów mogą wywołać skok aktywności [14]. To może skutkować dużą liczbą zapisów z użyciem tego samego klucza (gdy klucz to np. identyfikator celebryty lub identyfikator działań komentowanych przez użytkowników). Haszowanie klucza wtedy nie pomaga, ponieważ skróty dwóch identycznych identyfikatorów też są takie same.

Obecnie większość systemów danych nie potrafi automatycznie kompensować tego rodzaju wysocze asymetrycznego obciążenia roboczego, dlatego zadanie to musi wykonywać aplikacja. Na przykład jeśli wiadomo, że jeden klucz generuje wysokie obciążenie, prostą techniką stanowi dodawanie liczby losowej na początku lub na końcu tego klucza. Już dwucyfrowa liczba losowa (dziesiętna) rozdziela zapisy klucza równomiernie na 100 różnych kluczy, co pozwala rozmieścić je w różnych partycjach.

Jednak po podziale zapisów między różne klucze odczyty wymagają dodatkowej pracy, ponieważ trzeba wczytać i połączyć dane na podstawie wszystkich 100 kluczy. Ta technika wymaga też przechowywania dodatkowych informacji. Sensowne jest dołączanie liczb losowych tylko do niewielkiej liczby często używanych kluczy. W przypadku zdecydowanej większości kluczy generujących niewielką liczbę zapisów ta technika powoduje niepotrzebne koszty. Dlatego musisz śledzić, które klucze są rozdzielane.

Możliwe, że w przyszłości systemy danych będą potrafiły automatycznie wykrywać i kompensować asymetryczne obciążenie robocze. Jednak na razie musisz uwzględnić różne kompromisy we własnej aplikacji.

# Podział na partycje a indeksy pomocnicze

Systemy podziału na partycje opisane do tej pory są oparte na modelu klucz-wartość. Jeśli dostęp do rekordów odbywa się wyłącznie na podstawie klucza głównego, można ustalić partycję według klucza i na tej podstawie kierować żądania odczytu i zapisu do partycji odpowiedzialnej za dany klucz.

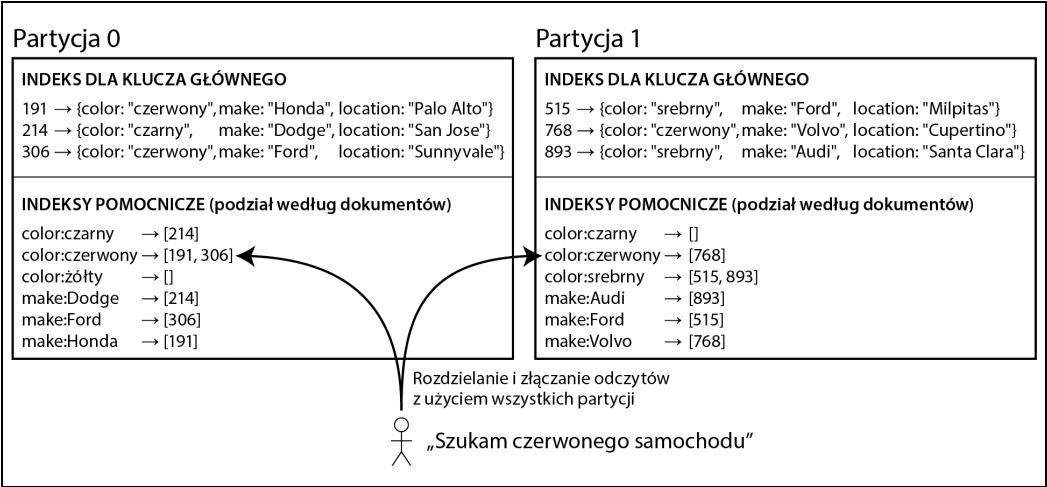
Sytuacja staje się bardziej skomplikowana, jeśli używane są indeksy pomocnicze (zob. też punkt „Inne struktury używane dla indeksów”). Indeks pomocniczy zwykle nie służy do unikatowego identyfikowania rekordów, a raczej do wyszukiwania wystąpień określonej wartości: znajdź wszystkie operacje użytkownika 123, znajdź wszystkie artykuły ze słowem bzdury, znajdź wszystkie samochody o kolorze czerwony itd.

Indeksy pomocnicze są powszechne w bazach relacyjnych i częste także w bazach opartych na dokumentach. W wielu bazach używających modelu klucz-wartość (np. HBase i Voldemort) unika się indeksów pomocniczych z powodu wzrostu złożoności implementacji. Jednak w innych bazach (takich jak Riak) zaczęto stosować te indeksy, ponieważ są bardzo przydatne do modelowania danych. Ponadto indeksy pomocnicze są istotą serwerów wyszukiwania takich jak Solr i Elasticsearch.

Problem z indeksami pomocniczymi polega na tym, że nie zapewniają eleganckiego odwzorowania na partycje. Istnieją dwa główne podejścia do podziału baz z indeksami pomocniczymi na partycje: na podstawie dokumentów i na podstawie pojęć.

## Podział indeksów pomocniczych na podstawie dokumentów

Wyobraź sobie np., że zarządzasz witryną do sprzedaży używanych samochodów (zob. rysunek 6.4). Każda oferta ma unikatowy identyfikator (nazwijmy go *identyfikatorem dokumentu*), a baza jest dzielona na partycje według tych identyfikatorów (np. identyfikatory od 0 do 499 znajdują się w partycji 0, identyfikatory od 500 do 999 w partycji 1 itd.).



Rysunek 6.4. Podział indeksów pomocniczych na podstawie dokumentów

Chcesz umożliwić użytkownikom wyszukiwanie samochodów oraz filtrowanie ich według kolorów i producenta. Potrzebujesz więc indeksów pomocniczych dla cech `color` i `make` (w bazie opartej na dokumentach będą to pola, a w bazie relacyjnej — kolumny). Jeśli zadeklarowałeś indeks, baza może indeksować dane automatycznie<sup>2</sup>. Na przykład za każdym razem, gdy do bazy dodawany jest czerwony samochód, partycja bazy automatycznie dodaje go do listy identyfikatorów dokumentów dla elementu `color:czerwony` w indeksie.

W tym sposobie indeksowania każda partycja jest całkowicie odrębna. Każda partycja obejmuje własne indeksy pomocnicze pokrywające wyłącznie dokumenty z danej partycji. Nie ma znaczenia, jakie dane są przechowywane w innych partycjach. W trakcie zapisu danych w bazie (w celu dodania, usunięcia lub zaktualizowania dokumentu) trzeba uwzględnić tylko partycję zawierającą identyfikator zapisywanego dokumentu. Dlatego indeks dzielony według dokumentów jest też nazywany *indeksem lokalnym* (w odróżnieniu od opisanego w następnym punkcie *indeksu globalnego*).

Jednak odczyt z indeksu podzielonego według dokumentów wymaga ostrożności. Jeśli nie zarządzasz identyfikatorami dokumentów w specjalny sposób, nie ma powodu, dla którego wszystkie samochody o określonym kolorze lub danej marki miałyby znajdować się w tej samej partycji. Na rysunku 6.4 czerwone samochody występują w partycjach 0 i 1. Dlatego jeśli chcesz znaleźć auta w tym kolorze, musisz przesłać zapytanie do *wszystkich* partycji i połączyć wszystkie otrzymane wyniki.

Tę metodę kierowania zapytań do baz podzielonych na partycje można nazwać *rozdzielaniem i łączaniem* (ang. *scatter/gather*). Może ona spowodować, że zapytania dotyczące odczytu z użyciem indeksów pomocniczych będą kosztowne. Nawet jeśli zapytanie do partycji zgłasza się równolegle, metoda rozdzielania i łączania jest narażona na zwiększenie liczby skrajnych wartości opóźnienia (zob. punkt „Percentyle w praktyce”). Mimo to powszechnie stosuje się tę technikę. W bazach MongoDB, Riak [15], Cassandra [16], Elasticsearch [17], SolrCloud [18] i VoltDB [19] używane są indeksy pomocnicze z podziałem według dokumentów. Producenci większości baz zalecają takie budowanie struktury schematu podziału na partycję, aby indeksy pomocnicze mogły być obsługiwane z poziomu jednej partycji. Jednak nie zawsze jest to możliwe — zwłaszcza gdy w jednym zapytaniu używa się kilku indeksów pomocniczych (np. w trakcie filtrowania samochodów jednocześnie według koloru i marki).

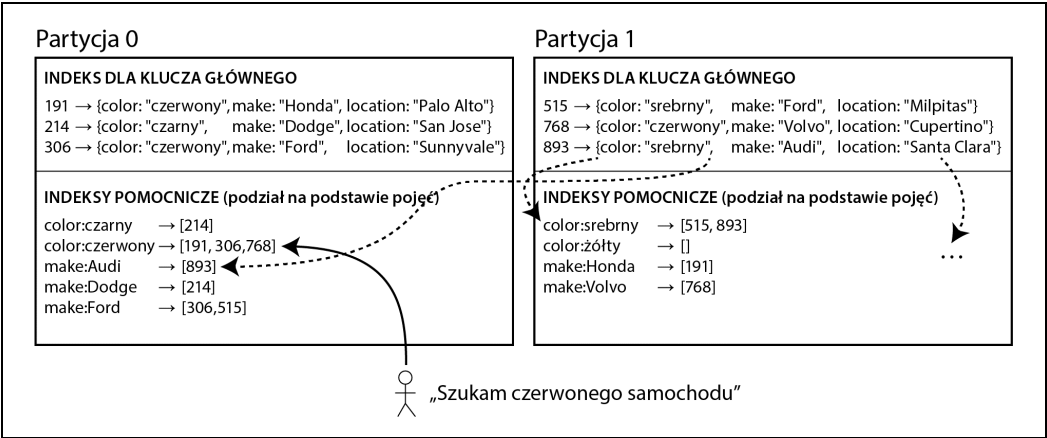
## Podział indeksów pomocniczych według pojęć

Zamiast używać dla każdej partycji odrębnego indeksu pomocniczego (*indeksu lokalnego*), można utworzyć *indeks globalny* pokrywający dane z wszystkich partycji. Nie wystarczy jednak zapisać tego indeksu w jednym węźle, ponieważ zapewne stałby się wtedy wąskim gardłem, co niweczy cel podziału na partycje. Indeks globalny też wymaga podziału na partycje, można go jednak podzielić inaczej niż indeks dla klucza głównego.

---

<sup>2</sup> Jeśli baza obsługuje tylko model klucz – wartość, może Cię kusić samodzielna implementacja indeksu pomocniczego przez utworzenie odwzorowania z wartości na identyfikatory dokumentu w kodzie aplikacji. Jeżeli wybierzesz to rozwiązanie, musisz zachować dużą ostrożność, aby zapewnić, że indeksy pozostaną spójne z danymi. Warunki wyścigu i nieregularne niepowodzenia zapisu (gdy pewne zmiany są zapisywane, a inne nie) mogą bardzo łatwo doprowadzić do braku synchronizacji danych (zob. punkt „Potrzeba obsługi transakcji z użyciem wielu obiektów”).

Na rysunku 6.5 pokazano, jak może wyglądać to rozwiązanie. Czerwone samochody z wszystkich partycji występują w indeksie w elemencie `color:czerwony`, jednak indeks został tak podzielony, że kolory zaczynające się na litery od *a* do *r* występują w partycji 0, a kolory od *s* do *z* w partycji 1. Indeks dotyczący marki samochodu został podzielony w podobny sposób (granice między partycjami wyznaczają litery *f* i *h*).



Rysunek 6.5. Podział indeksów pomocniczych według pojęć

Ten indeks nazywa się *podzielonym według pojęć*, ponieważ to szukane pojęcie określa partycję indeksu. Tu pojęciem jest np. `color:czerwony`. Określenie *pojęcie* pochodzi z obszaru indeksów pełnotekstowych (to specjalny rodzaj indeksów pomocniczych), gdzie pojęciami są wszystkie występujące w dokumencie słowa.

Podobnie jak wcześniej można podzielić indeks tylko według pojęcia lub na podstawie skrótu pojęcia. Podział według samego pojęcia może być przydatny do skanowania przedziałów (np. według cech liczbowych takich jak cena ofertowa samochodu), natomiast podział na podstawie skrótu zapewnia bardziej równomierny rozkład obciążenia.

Zaletę indeksu globalnego (z podziałem według pojęć) w porównaniu z indeksem dzielonym według dokumentów stanowi to, że umożliwia wydajniejsze odczyty. Zamiast stosować rozdzielanie i złączanie z użyciem wszystkich partycji, klient musi jedynie zgłosić żądanie do partycji zawierającej szukane pojęcie. Wadą indeksu globalnego są wolniejsze i bardziej skomplikowane zapisy, ponieważ zapis w jednym dokumencie może wtedy wpływać na wiele partycji indeksu (każde pojęcie z dokumentu może się znajdować w innej partycji z innego węzła).

W idealnym świecie indeks zawsze jest aktualny, a każdy dokument zapisywany w bazie zostaje natychmiast odzwierciedlony w indeksie. Jednak dla indeksu z podziałem według pojęć wymagałoby to transakcji rozproszonych z uwzględnieniem wszystkich partycji, których dotyczył zapis. Nie wszystkie bazy obsługują taką możliwość (zob. rozdziały 7. i 9.).

W praktyce aktualizacje globalnych indeksów pomocniczych są często asynchroniczne (oznacza to, że jeśli wczytasz indeks krótko po zapisie, wprowadzona zmiana może nie być jeszcze w nim widoczna). Na przykład w bazie DynamoDB Amazonu globalne indeksy pomocnicze są w normalnych warunkach aktualizowane w ułamku sekundy, jednak usterki w infrastrukturze mogą powodować dłuższe opóźnienia w przekazywaniu danych [20].

Inne zastosowania globalnych indeksów dzielonych na podstawie pojęć to wyszukiwarka w bazie Riak [21] i hurtownia danych Oracle, gdzie możliwy jest wybór indeksowania lokalnego i globalnego [22]. Do tematu implementowania indeksów pomocniczych dzielonych na podstawie pojęć wrócimy w rozdziale 12.

## Równoważenie partycji

Z czasem różne rzeczy w bazie się zmieniają:

- Rośnie przepustowość zapytań, dlatego warto dodać więcej procesorów w celu obsługi obciążenia.
- Rośnie wielkość zbioru danych, dlatego warto dodać nowe dyski i pamięć RAM na potrzeby przechowywania go.
- Maszyny się psują, a inne muszą przejmować ich zadania.

Wszystkie te zmiany wymagają przenoszenia danych i żądań z jednego węzła do innego. Proces przenoszenia obciążenia z jednego węzła w klastrze do innego to *równoważenie*.

Niezależnie od używanego schematu podziału na partycje równoważenie zwykle powinno spełniać pewne minimalne wymagania:

- Po równoważeniu obciążenie (dane, żądania odczytu i zapisu) powinno być równo rozdzielone między węzły klastra.
- W trakcie równoważenia baza powinna stale akceptować odczyty i zapisy.
- Między węzłami należy przenosić nie więcej niż niezbędną ilość danych, aby umożliwić szybkie równoważenie i zminimalizować obciążenie sieci oraz liczbę operacji wejścia-wyjścia na dysku.

## Strategie równoważenia

Istnieje kilka różnych sposobów przypisywania partycji do węzłów [23]. Omówmy pokrótce każdy z nich.

### Jak tego nie robić — skrót dzielony modulo przez $N$

Gdy dane są dzielone według skrótu klucza, to — jak opisano wcześniej (rysunek 6.3) — najlepiej rozdzielić możliwe skróty na przedziały i przypisać każdy przedział do partycji (np. przypisywać *klucz* do partycji 0, jeśli  $0 \leq \text{skrót}(\text{klucz}) < b_0$ , do partycji 1, jeśli  $b_0 \leq \text{skrót}(\text{klucz}) < b_1$  itd.).

Może się zastanawiałeś, dlaczego nie stosować po prostu dzielenia modulo (w wielu językach programowania odpowiada mu operator %). Na przykład operacja *skrót(klucz) modulo 10* daje liczbę z przedziału od 0 do 9 (jeśli zapiszesz skrót jako liczbę dziesiętną, *skrót modulo 10* będzie ostatnią cyfrą skrótu). Jeśli używasz 10 węzłów ponumerowanych od 0 do 9, wydaje się, że to łatwy sposób na przypisanie każdego klucza do węzła.

Problem z techniką *modulo  $N$*  polega na tym, że jeśli liczba węzłów  $N$  się zmienia, większość kluczy trzeba będzie przenieść z jednego węzła do innego. Załóżmy, że  $\text{skrót}(\text{klucz}) = 123456$ . Jeżeli początkowo jest 10 węzłów, ten klucz trafia do węzła 6 (ponieważ  $123456 \text{ modulo } 10 = 6$ ). Gdy liczba węzłów wzrośnie do 11, klucz trzeba przenieść do węzła 3 ( $123456 \text{ modulo } 11 = 3$ ), a dla 12 wę-

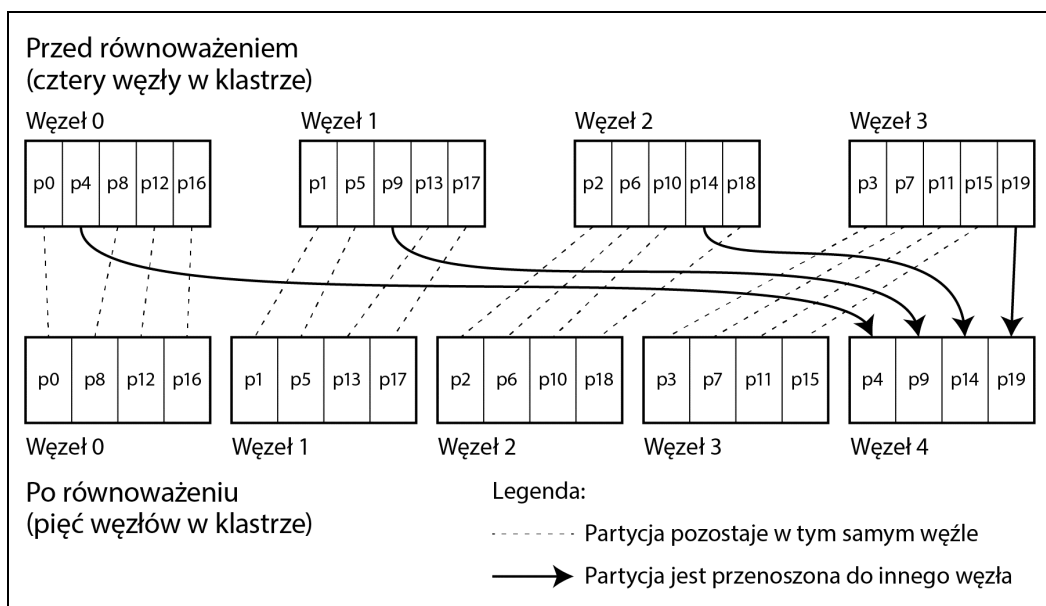
złów klucz należy umieścić w węźle 0 ( $123456 \bmod 12 = 0$ ). Tak częste przenoszenie danych sprawia, że równoważenie jest nadmiernie kosztowne.

Potrzeba podejścia, które nie powoduje przenoszenia danych częściej niż to konieczne.

### Stała liczba partycji

Na szczęście istnieje stosunkowo proste rozwiązanie — należy utworzyć znacznie więcej partycji niż węzłów i przypisać kilka partycji do każdego węzła. Na przykład bazę działającą w klastrze z 10 węzłami można od samego początku rozbić na 1000 partycji, tak aby do każdego węzła przypisanych było ok. 100 partycji.

Jeśli teraz węzeł jest dodawany do klastra, może *ukraść* kilka partycji z każdego istniejącego węzła, tak aby partycje ponownie były równomiernie rozdzielone. Ten proces pokazano na rysunku 6.6. Po usunięciu węzła z klastra następuje odwrotny proces.



Rysunek 6.6. Dodawanie nowego węzła do klastra bazodanowego z wieloma partycjami na węzeł

Miedzy węzłami przenoszone są tylko całe partycje. Liczba partycji się nie zmienia podobnie jak przypisanie kluczy do partycji. Jedyną rzeczą, jaka jest modyfikowana, to przydział partycji do węzłów. Ta zmiana nie odbywa się natychmiastowo. Potrzeba czasu, aby przesłać dużą ilość danych w sieci. Dlatego dla odczytów i zapisów odbywających się w trakcie przesyłania danych stosuje się dawny podział partycji.

Teoretycznie można nawet uwzględnić różnice w wydajności sprzętu w klastrze. Dzięki przypisaniu większej liczby partycji do węzłów o większych możliwościach można wymusić, by te węzły przejmowały większą część obciążenia.

Ten model równoważenia stosuje się w bazach Riak [15], Elasticsearch [24], Couchbase [10] i Voldemort [25].

W opisanej konfiguracji liczba partycji jest zwykle ustalana w momencie konfigurowania bazy i później się nie zmienia. Choć teoretycznie możliwe jest dzielenie i scalanie partycji (zob. następny punkt), stała liczba partycji ułatwia eksploatację systemu. Dlatego w wielu bazach o niezmienniej liczbie partycji nie stosuje się podziału partycji. Tak więc skonfigurowana na początku liczba partycji wyznacza maksymalną liczbę węzłów. Z tego powodu musi być ona wystarczająco wysoka, by uwzględnić przyszły wzrost. Jednak z zarządzaniem każdą partycją związane są koszty, dlatego ustalenie zbyt wysokiej liczby jest szkodliwe.

Dobór odpowiedniej liczby partycji jest trudny, gdy łączna wielkość zbioru danych zmienia się w dużym stopniu (np. początkowo jest niewielka, jednak później znacznie rośnie). Ponieważ każda partycja obejmuje stałą część wszystkich danych, wielkość każdej partycji rośnie proporcjonalnie do łącznej ilości danych w klastrze. Gdy partycje są bardzo duże, równoważenie i odzyskiwanie danych po awarii węzła staje się kosztowne. Jeśli jednak partycje są zbyt małe, generują za wysokie koszty. Najlepszą wydajność można uzyskać, gdy wielkość partycji jest „w sam raz” — ani za duża, ani za mała. Jej określenie może być trudne, jeśli liczba partycji jest stała, natomiast wielkość zbioru danych się zmienia.

### **Dynamiczny podział na partycje**

W bazach, w których stosuje się podział z użyciem przedziałów kluczy (zob. punkt „Podział na podstawie przedziałów kluczy”), stała liczba partycji z ustalonymi granicami byłaby bardzo niewygodna. Jeśli granice są błędnie określone, wszystkie dane mogą trafić do jednej partycji, a wszystkie pozostałe będą puste. Ręczna zmiana konfiguracji granic partycji jest bardzo żmudna.

Z tego powodu bazy dzielone według przedziałów kluczy (np. HBase i RethinkDB) tworzą partycje dynamicznie. Gdy partycja się rozrasta i przekracza skonfigurowany rozmiar (w bazie HBase domyślnie wynosi on 10 GB), następuje jej podział na dwie partycje, tak by każda z nich zawierała mniej więcej połowę danych [26]. Z drugiej strony gdy usuniętych zostanie dużo danych i partycja zmaleje poniżej ustalonego progu, może zostać scalona z przyległą partycją. Ten proces przypomina to, co się dzieje na najwyższym poziomie b-drzewa (zob. punkt „B-drzewa”).

Każda partycja jest przypisywana do jednego węzła, a każdy węzeł może obsługiwać wiele partycji (podobnie jak wtedy, gdy liczba partycji jest stała). Po podziale dużej partycji jedna z jej dwóch połówek może zostać przeniesiona do innego węzła, by zrównoważyć obciążenie. W bazie HBase przenoszenie plików partycji odbywa się z użyciem systemu HDFS (używanego w niej rozproszonego systemu plików) [3].

Zaletą dynamicznego podziału na partycje stanowi to, że liczba partycji dostosowuje się do łącznej ilości danych. Jeśli ilość danych jest niewielka, wystarczy mała liczba partycji, dlatego koszty są niskie. Gdy ilość danych jest duża, wielkość każdej partycji jest ograniczona do konfigurowalnego maksimum [23].

Wadą jest to, że pusta baza zawiera jedną partycję, ponieważ nie ma określonych z góry informacji o tym, gdzie wyznaczyć granice partycji. Gdy zbiór danych jest niewielki (do momentu podziału pierwszej partycji), wszystkie zapisy muszą być przetwarzane przez jeden węzeł, podczas gdy inne pozostają bezczynne. Aby ograniczyć ten problem, w bazach HBase i MongoDB można skonfigurować w pustej bazie początkowy zestaw partycji (to tzw. *podział wstępny*). Przy podziale na podstawie przedziałów kluczy podział wstępny wymaga znajomości rozkładu kluczy [4, 26].



Dynamiczny podział na partycje jest odpowiedni nie tylko dla danych dzielonych według przedziałów kluczy, ale też dla danych dzielonych według skrótów. W bazie MongoDB od wersji 2.4 obsługiwany jest podział zarówno według przedziałów kluczy, jak i według skrótów. W obu podejściach partycje są tworzone dynamicznie.

### **Podział na partycje proporcjonalnie do liczby węzłów**

W dynamicznym podziale na partycje ich liczba jest proporcjonalna do wielkości zbioru danych, ponieważ procesy podziału i scalania utrzymują wielkość każdej partycji między ustalonym minimum i maksimum. Z kolei gdy liczba partycji jest stała, wielkość każdej z nich jest proporcjonalna do wielkości zbioru danych. W obu sytuacjach liczba partycji nie zależy od liczby węzłów.

Trzecia możliwość, stosowana w bazach Cassandra i Ketama, to określanie liczby partycji proporcjonalnie do liczby węzłów. Oznacza to stałą liczbę partycji *na węzeł* [23, 27, 28]. W takiej sytuacji wielkość każdej partycji rośnie proporcjonalnie do wielkości zbioru danych, gdy liczba węzłów się nie zmienia, natomiast po zwiększeniu liczby węzłów partycje znów stają się mniejsze. Ponieważ większa ilość danych zwykle wymaga większej liczby węzłów, to podejście pozwala zachować stosunkowo stabilną wielkość każdej partycji.

Gdy nowy węzeł dołącza do klastra, losowo wybiera stałą liczbę istniejących partycji do podziału i staje się właścicielem połowy każdej z podzielonych partycji (drugie połowy pozostają w dawnym miejscu). Losowość może prowadzić do nierównych podziałów, jednak przy uśrednieniu dla dużej liczby partycji (w bazie Cassandra domyślnie wynosi ona 256 partycji na węzeł) nowy węzeł przejmuje uczciwie wyznaczoną część obciążenia od istniejących węzłów. W bazie Cassandra 3.0 wprowadzono nowy algorytm równoważenia pozwalający uniknąć nierównych podziałów [29].

Losowy dobór granic partycji wymaga stosowania podziału na podstawie skrótów (tak by granice można było ustalić dla przedziału liczb zwracanych przez funkcję skrótu). To podejście jest najbardziej zbliżone do pierwotnej definicji spójnego haszowania [7] (zob. punkt „Spójne haszowanie”). Nowsze funkcje skrótu pozwalają uzyskać podobny efekt z mniejszymi kosztami związanymi z metadanymi [8].

## **Eksploatacja — równoważenie automatyczne lub ręczne**

Z równoważeniem wiąże się jedno ważne pytanie, które zostało pominięte: czy równoważenie odbywa się automatycznie, czy ręcznie?

Istnieje spektrum możliwości między w pełni automatycznym równoważeniem (system automatycznie i bez żadnej interakcji z administratorem decyduje, kiedy przenieść partycje z jednego węzła do innego) a podejściem w pełni ręcznym (przydział partycji do węzłów jest bezpośrednio konfigurowany przez administratora i zmienia się tylko wtedy, gdy administrator zmieni konfigurację). Na przykład bazy Couchbase, Riak i Voldemort automatycznie generują sugerowany przydział partycji, jednak przed jego wprowadzeniem wymagają, by administrator go zatwierdził.

W pełni automatyczne równoważenie może być wygodne, ponieważ wymaga mniej pracy operacyjnej w ramach codziennej konserwacji systemu. To rozwiązanie może być jednak nieprzewidywalne. Równoważenie to kosztowna operacja, ponieważ wymaga przekierowywania żądań i prze-

noszenia dużych ilości danych z jednego węzła na inny. Jeśli nie zostanie zachowana ostrożność, ten proces może przeciążyć sieć lub węzły i spowolnić obsługę innych żądań w trakcie równoważenia.

Automatyzacja może być niebezpieczna w połączeniu z automatycznym wykrywaniem awarii. Załóżmy, że jeden węzeł jest przeciążony i tymczasowo wolno reaguje na żądania. Inne węzły uznają, że przeciążona jednostka przestała działać, co skutkuje automatycznym równoważeniem klastra i przeniesieniem obciążenia z powolnego węzła. To dodatkowo obciąża przeciążony węzeł, inne węzły i sieć. Pogarsza to sytuację i może prowadzić do awarii kaskadowej.

Z tego powodu dobrze jest umieścić człowieka w pętli równoważenia. Proces jest wtedy wolniejszy niż przy pełnej automatyzacji, człowiek może jednak pomóc zapobiec operacyjnym niespodziankom.

## Trasowanie żądań

Zbiór danych jest już podzielony na partycje w wielu węzłach z różnych maszyn. Jednak jedno pytanie pozostaje otwarte: gdy klient chce zgłosić żądanie, skąd wie, z którym węzłem ma się połączyć? W trakcie równoważenia partycji ich przydział do węzłów się zmienia. Coś musi śledzić te zmiany, aby móc odpowiedzieć na pytanie: „Gdy chcę wczytać lub zapisać wartość klucza *foo*, z którym adresem IP i numerem portu mam się połączyć?”.

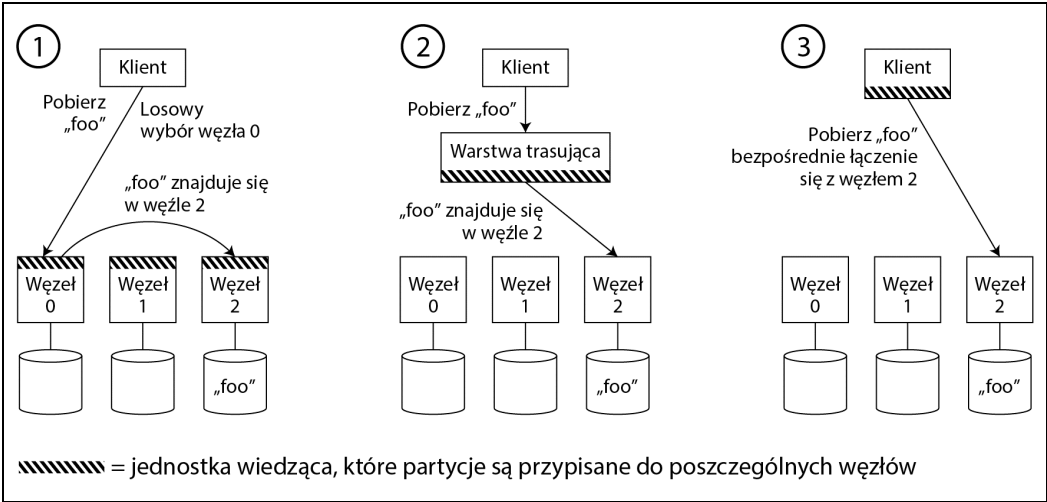
To przykład ogólniejszego problemu *wykrywania usług*, który dotyczy nie tylko baz danych. Ten problem występuje w każdym oprogramowaniu dostępnym przez sieć, zwłaszcza gdy cel stanowi wysoka dostępność (czyli gdy system działa z nadmiarowymi maszynami). Wiele firm opracowało własne wewnętrzne narzędzia do wykrywania usług. Liczne z tych narzędzi zostały udostępnione jako otwarte oprogramowanie [30].

Na ogólnym poziomie istnieje kilka różnych rozwiązań tego problemu (przedstawia je rysunek 6.7):

1. Umożliwianie klientom kontaktu z dowolnym węzłem (np. za pomocą cyklicznego równoważnika obciążenia). Jeśli dany węzeł przypadkowo przechowuje partycję, której dotyczy żądanie, może obsłużyć je bezpośrednio. W przeciwnym razie przekazuje żądanie do odpowiedniego węzła, otrzymuje odpowiedź i przesyła ją do klienta.
2. Przesyłanie wszystkich żądań od klientów najpierw do warstwy trasującej. Określa węzeł, który powinien obsłużyć każde żądanie, i odpowiednio przekazuje to żądanie. Warstwa trasująca sama nie obsługuje żadnych żądań. Działa tylko jak równoważnik obciążenia mający wiedzę o partycjach.
3. Wymaganie, by klienci znały podział na partycje i wiedziały, jak są one przypisane do węzłów. Wtedy klient może się bezpośrednio połączyć z odpowiednim węzłem (bez pośrednika).

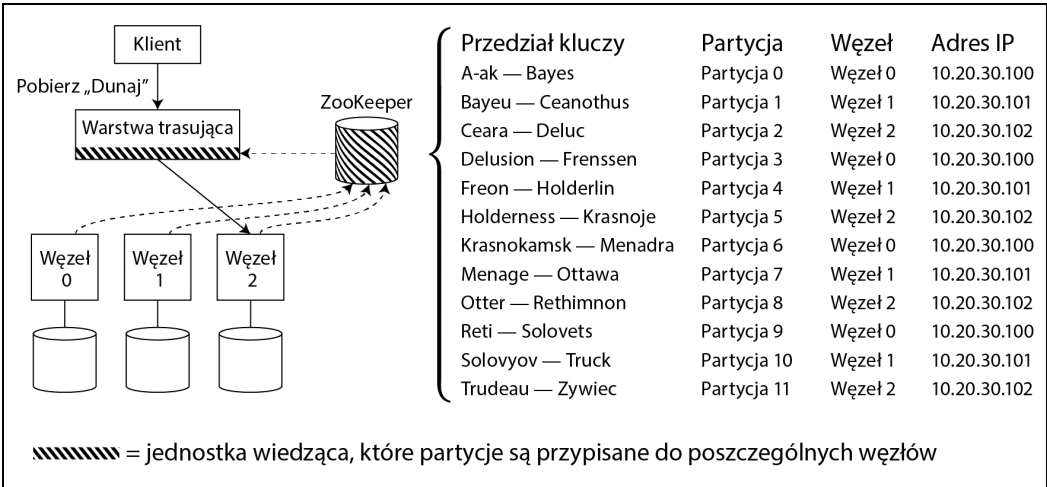
We wszystkich sytuacjach najważniejszy problem to, skąd komponent podejmujący decyzje o trasowaniu (może nim być jeden z węzłów, warstwa trasująca lub klient) ma wiedzieć o zmianach w przypisaniu partycji do węzłów?

To trudny problem, ponieważ ważne jest, by wszystkie jednostki miały spójny obraz. W przeciwnym razie żądania trafiają do niewłaściwych węzłów i nie zostaną poprawnie obsłużone. Istnieją protokoły osiągnięcia konsensusu w systemach rozproszonych, jednak ich poprawna implementacja jest skomplikowana (zob. rozdział 9.).



Rysunek 6.7. Trzy różne sposoby trasowania żądań do odpowiedniego węzła

Wiele rozproszonych systemów danych wykorzystuje do śledzenia metadanych w klastrze odrębną usługę koordynującą taką jak ZooKeeper. Takie rozwiązanie ilustruje rysunek 6.8. Każdy węzeł rejestruje się w ZooKeeperze, a ten przechowuje miarodajne odwzorowanie partycji na węzły. Inne jednostki, np. warstwa trasująca lub klient potrzebujący wiedzy o partycjach, mogą subskrybować te informacje w ZooKeeperze. Gdy właściciel partycji się zmienia albo węzeł zostaje dodany lub usunięty, ZooKeeper powiadamia o tym warstwę trasującą, by zawierała aktualne informacje.



Rysunek 6.8. Używanie ZooKeepera do śledzenia przypisania partycji do węzłów

Na przykład w bazie Espresso w serwisie LinkedIn do zarządzania klastrem używany jest Helix [31] (który z kolei oparty jest na ZooKeeperze) tworzący warstwę trasującą działającą tak jak pokazano na rysunku 6.8. Bazy HBase, SolrCloud i Kafka także używają ZooKeepera do śledzenia przypisania partycji. W MongoDB architektura jest podobna, jednak tam używa się niestandardowej implementacji serwera konfiguracyjnego, a warstwę trasującą tworzą demony *mongo*.

W bazach Cassandra i Riak wykorzystuje się inne podejście. Tam węzły używają *protokołu gossip* do rozpowszechniania zmian w stanie klastra. Żądania można przysyłać do dowolnego węzła, który je przekazuje do odpowiedniego węzła z żadaną partycją (podejście 1. z rysunku 6.7). Ten model zwiększa złożoność węzłów z bazą danych, jednak pozwala uniknąć zależności od zewnętrznej usługi koordynującej takiej jak ZooKeeper.

Baza Couchbase nie zapewnia automatycznego równoważenia, co upraszcza projekt. Standardowo jest konfigurowana z użyciem warstwy trasującej nazywanej *moxi*, która pobiera informacje o zmianach w trasowaniu od węzłów klastra [32].

Gdy używasz warstwy trasującej lub przesyłasz żądania do losowego węzła, klienci nadal muszą znaleźć adres IP, z którym mają się połączyć. Zmiany tych adresów nie są równie szybkie jak zmiany przypisań partycji do węzłów, dlatego wystarczy wykorzystać system DNS.

## Równoległe wykonywanie zapytań

Do tej pory uwzględniane były bardzo proste zapytania wczytujące lub zapisujące jeden klucz (a także zapytania z rozdzielaniem i łączeniem dla indeksów pomocniczych dzielonych według dokumentów). Jest to w przybliżeniu poziom dostępu obsługiwany przez większość rozproszonych baz NoSQL.

Jednak relacyjne bazy typu **MPP** (ang. *massively parallel processing*), często stosowane do analityki, są znacznie bardziej zaawansowane, jeśli chodzi o obsługiwane typy zapytań. Typowe zapytanie w hurtowni danych obejmuje kilka złączeń, filtrowanie, grupowanie i operacje agregacji. Optymalizator zapytań w bazach MPP rozбивa takie złożone zapytanie na etapy wykonywania i partycje. Wiele tych etapów może być wykonywanych równoległe w różnych węzłach klastra bazodanowego. Równoległe wykonywanie jest korzystne zwłaszcza w zapytaniach obejmujących skanowanie dużych części zbioru danych.

Szybkie równoległe wykonywanie zapytań w hurtowni danych to specjalistyczne zagadnienie, które z powodu znaczenia biznesowego analityki cieszy się dużym zainteresowaniem firm komercyjnych. Niektóre techniki równoległego wykonywania zapytań są opisane w rozdziale 10. Szczegółowe omówienie technik używanych w bazach z równoległym przetwarzaniem znajdziesz w literaturze [1, 33].

## Podsumowanie

W tym rozdziale opisano różne sposoby podziału dużych zbiorów danych na mniejsze podzbiory. Podział na partycje jest konieczny, gdy ilość danych jest tak duża, że ich przechowywanie i przetwarzanie na jednej maszynie staje się niewykonalne.

Cel podziału na partycje to równomierny podział danych i związanego z zapytaniami obciążenia między wiele maszyn. Należy przy tym unikać powstawania hot spotów (węzłów o nieproporcjonalnie wysokim obciążeniu). Wymaga to doboru schematu podziału odpowiedniego dla danych i równoważenia partycji, gdy węzły są dodawane do klastra lub z niego usuwane.

Opisano tu dwa główne podejścia do podziału na partycje:

- *Podział według przedziałów kluczy*, gdzie klucze są sortowane, a partycja obejmuje wszystkie klucze od określonej wartości minimalnej do ustalonego maksimum. Sortowanie ma tę zaletę, że możliwe są wydajne zapytania o przedziały. Jednak pojawia się wtedy ryzyko wystąpienia hot spotów, jeśli aplikacja często używa kluczy, które po posortowaniu znajdują się blisko siebie.

W tym podejściu partycje są zwykle dynamicznie równoważone przez podział przedziału na dwa podprzedziały, gdy partycja staje się zbyt duża.

- *Podział z użyciem skrótów*, gdzie do każdego klucza stosowana jest funkcja skrótu, a partycja obejmuje przedział skrótów. Ta metoda narusza uporządkowanie kluczy, przez co zapytania o przedziały stają się niewydajne, jednak może skutkować bardziej równomiernym rozkładem obciążenia.

Gdy dzielisz dane z użyciem skrótów, często z góry tworzona jest stała liczba partycji, co pozwala przypisać kilka partycji do każdego węzła i przenosić całe partycje między węzłami, gdy te ostatnie są dodawane lub usuwane. Można się też posłużyć dynamicznym podziałem na partycje.

Możliwe są także podejścia hybrydowe, np. z wykorzystaniem klucza złożonego. Jedna część klucza służy wtedy do identyfikowania partycji, a inna jest przeznaczona do sortowania danych.

Omówiono tu również zależności między podziałem na partycje a indeksami pomocniczymi. Indeks pomocniczy też trzeba dzielić. Używa się dwóch metod:

- *Indeksy dzielone według dokumentów* (indeksy lokalne). Indeksy pomocnicze są wtedy przechowywane w tej samej partycji co klucz główny i wartość. To oznacza, że przy zapisie konieczna jest aktualizacja tylko jednej partycji, jednak odczyt indeksu pomocniczego wymaga rozdzielania i łączenia z użyciem wszystkich partycji.
- *Indeksy dzielone na podstawie pojęć* (indeksy globalne). Indeksy pomocnicze są wtedy dzielone na partycje pojedynczo na podstawie indeksowanych wartości. Wpis w indeksie pomocniczym może obejmować rekordy z wszystkich partycji klucza głównego. Gdy dokument jest zapisywany, czasem trzeba zaktualizować kilka partycji z indeksem pomocniczym. Jednak odczyt można obsłużyć za pomocą jednej partycji.

W końcowej części omówiono techniki kierowania zapytań do odpowiedniej partycji. Stosuje się różne rozwiązania: od prostego równoważenia obciążenia z uwzględnieniem partycji po zaawansowane systemy równoległego wykonywania zapytań.

Każda partycja z założenia działa w dużym stopniu niezależnie. Umożliwia to skalowanie dzięki podziałowi bazy na wiele maszyn. Jednak czasem trudno jest analizować operacje, które muszą zapisywać dane w kilku partycjach. Co się np. stanie, jeśli zapis w jednej partycji zakończy się powodzeniem, a w innej — porażką? Odpowiedź na to pytanie znajdziesz w dalszych rozdziałach.

## Literatura cytowana

- [1] David J. DeWitt i Jim N. Gray, *Parallel Database Systems: The Future of High Performance Database Systems*, „Communications of the ACM”, rocznik 35, nr 6, s. 85 – 98, czerwiec 1992 (<http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/dewittgray92.pdf>; <https://dl.acm.org/citation.cfm?doid=129888.129894>).
- [2] Lars George, *HBase vs. BigTable Comparison*, [larsgeorge.com](http://www.larsgeorge.com), listopad 2009 (<http://www.larsgeorge.com/2009/11/hbase-vs-bigtable-comparison.html>).
- [3] *The Apache HBase Reference Guide*, Apache Software Foundation, [hbase.apache.org](http://hbase.apache.org), 2014 (<https://hbase.apache.org/book.html#book>).
- [4] MongoDB, Inc., *New Hash-Based Sharding Feature in MongoDB 2.4*, [blog.mongodb.org](http://blog.mongodb.org), 10 kwietnia 2013 (<https://www.mongodb.com/blog/post/new-hash-based-sharding-feature-in-mongodb-24>).
- [5] Ikai Lan, *App Engine Datastore Tip: Monotonically Increasing Values Are Bad*, [ikaisays.com](http://ikaisays.com), 25 stycznia 2011 (<https://ikaisays.com/2011/01/25/app-engine-datastore-tip-monotonically-increasing-values-are-bad/>).
- [6] Martin Kleppmann, *Java's hashCode Is Not Safe for Distributed Systems*, [martin.kleppmann.com](http://martin.kleppmann.com), 18 czerwca 2012 (<http://martin.kleppmann.com/2012/06/18/java-hashcode-unsafe-for-distributed-systems.html>).
- [7] David Karger, Eric Lehman, Tom Leighton i in., *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, w: „29th Annual ACM Symposium on Theory of Computing” (STOC), s. 654 – 663, 1997 (<https://www.akamai.com/us/en/multimedia/documents/technical-publication/consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-web-technical-publication.pdf>; <https://dl.acm.org/citation.cfm?doid=258533.258660>).
- [8] John Lamping i Eric Veach, *A Fast, Minimal Memory, Consistent Hash Algorithm*, [arxiv.org](http://arxiv.org), czerwiec 2014 (<https://arxiv.org/ftp/arxiv/papers/1406/1406.2294.pdf>).
- [9] Eric Redmond, *A Little Riak Book*, Version 1.4.0, Basho Technologies, wrzesień 2013 (<http://basho.com/posts/business/download-a-little-riak-book/>).
- [10] *Couchbase 2.5 Administrator Guide*, Couchbase, Inc., 2014 (<https://www.couchbase.com/customers>).
- [11] Avinash Lakshman i Prashant Malik, *Cassandra — A Decentralized Structured Storage System*, w: „3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware” (LADIS), październik 2009 (<http://www.cs.cornell.edu/Projects/ladis2009/papers/Lakshman-ladis2009.PDF>).
- [12] Jonathan Ellis, *Facebook's Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0*, [datastax.com](http://datastax.com), 12 września 2013 (<http://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html>).

- [13] *Introduction to Cassandra Query Language*, DataStax, Inc., 2014 ([http://docs.datastax.com/en/cql/3.1/cql/cql\\_intro\\_c.html](http://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html)).
- [14] Samuel Axon, *3% of Twitter's Servers Dedicated to Justin Bieber*, *mashable.com*, 7 września 2010 (<http://mashable.com/2010/09/07/justin-bieber-twitter/#9lFfUKsZzOqp>).
- [15] *Riak 1.4.8 Docs*, Basho Technologies, Inc., 2014 (<http://docs.basho.com/riak/1.4.8/>).
- [16] Richard Low, *The Sweet Spot for Cassandra Secondary Indexing*, *wentnet.com*, 21 października 2013 (<http://www.wentnet.com/blog/?p=77>).
- [17] Zachary Tong, *Customizing Your Document Routing*, *elasticsearch.org*, 3 czerwca 2013 (<https://www.elastic.co/blog/customizing-your-document-routing>).
- [18] *Apache Solr Reference Guide*, Apache Software Foundation, 2014 ([https://lucene.apache.org/solr/guide/6\\_6/](https://lucene.apache.org/solr/guide/6_6/)).
- [19] Andrew Pavlo, *H-Store Frequently Asked Questions*, *hstore.cs.brown.edu*, październik 2013 (<http://hstore.cs.brown.edu/documentation/faq/>).
- [20] *Amazon DynamoDB Developer Guide*, Amazon Web Services, Inc., 2014 (<http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/>).
- [21] Rusty Klophaus, *Difference Between 2I and Search*, e-mail na liście dyskusyjnej *riak-users*, *lists.basho.com*, 25 października 2011 ([http://lists.basho.com/pipermail/riak-users\\_lists.basho.com/2011-October/006220.html](http://lists.basho.com/pipermail/riak-users_lists.basho.com/2011-October/006220.html)).
- [22] Donald K. Burleson, *Object Partitioning in Oracle*, *dba-oracle.com*, 8 listopada 2000 ([http://www.dba-oracle.com/art\\_partit.htm](http://www.dba-oracle.com/art_partit.htm)).
- [23] Eric Evans, *Rethinking Topology in Cassandra*, w: *ApacheCon Europe*, listopad 2012 (<https://www.slideshare.net/jericevans/virtual-nodes-rethinking-topology-in-cassandra>).
- [24] Rafał Kuć, *Reroute API Explained*, *elasticsearchserverbook.com*, 30 września 2013 (<http://elasticsearchserverbook.com/reroute-api-explained/>).
- [25] *Project Voldemort Documentation*, *project-voldemort.com* (<http://www.project-voldemort.com/voldemort/>).
- [26] Enis Soztutar, *Apache HBase Region Splitting and Merging*, *hortonworks.com*, 1 lutego 2013 (<https://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/>).
- [27] Brandon Williams, *Virtual Nodes in Cassandra 1.2*, *datastax.com*, 4 grudnia 2012 (<https://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2>).
- [28] Richard Jones, *libketama: Consistent Hashing Library for Memcached Clients*, *metabrew.com*, 10 kwietnia 2007 (<https://www.metabrew.com/article/libketama-consistent-hashing-algo-memcached-clients>).
- [29] Branimir Lambov, *New Token Allocation Algorithm in Cassandra 3.0*, *datastax.com*, 28 stycznia 2016 (<https://www.datastax.com/dev/blog/token-allocation-algorithm>).

[30] Jason Wilder, *Open-Source Service Discovery*, [jasonwilder.com](http://jasonwilder.com), luty 2014 (<http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/>).

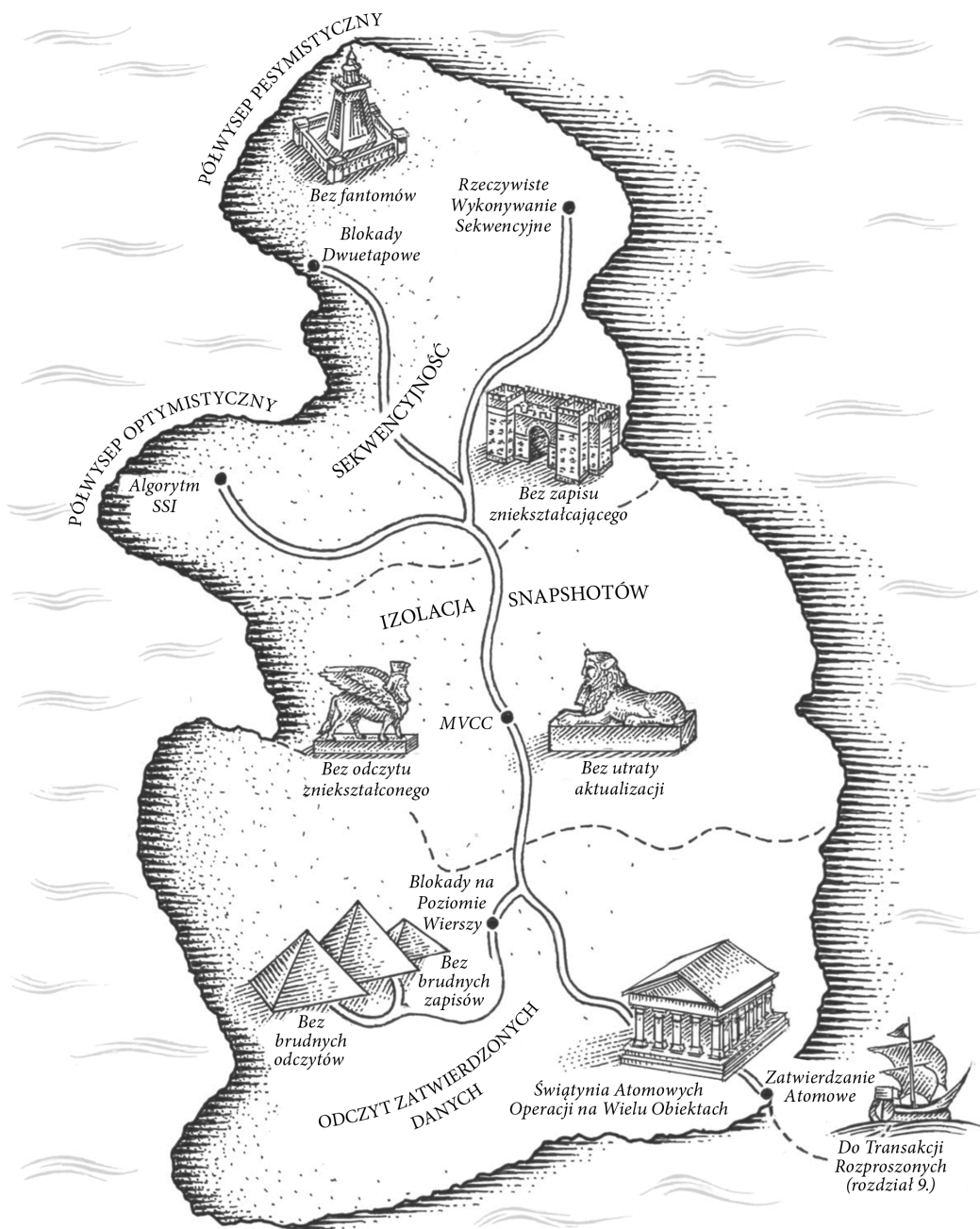
[31] Kishore Gopalakrishna, Shi Lu, Zhen Zhang i in., *Untangling Cluster Management with Helix*, w: „ACM Symposium on Cloud Computing” (SoCC), październik 2012 ([https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/helix\\_onecol.pdf?attachauth=ANoY7cpWUGQIECnt7rXyYRfTC0Ur1bbB2Ad77JIRd1wjSPkv9gEDrJdZ32HDkK\\_x7k4UItYWdUYhBtqOsy7xMXyvY4Vd5nBqdMfSKZift6\\_yK9jmAbQAH21SO8ZdKM5CDvZb-LzuTjNGiTr095jyhN89iydsNtc4DlozubDgoYtuC16EzQgVpEMLt6WQu5FI2up3xGVI47KV3OpoRAAMqn8b0STmco3GWw%3D%3D&attredirects=0;https://dl.acm.org/citation.cfm?doid=2391229.2391248](https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/helix_onecol.pdf?attachauth=ANoY7cpWUGQIECnt7rXyYRfTC0Ur1bbB2Ad77JIRd1wjSPkv9gEDrJdZ32HDkK_x7k4UItYWdUYhBtqOsy7xMXyvY4Vd5nBqdMfSKZift6_yK9jmAbQAH21SO8ZdKM5CDvZb-LzuTjNGiTr095jyhN89iydsNtc4DlozubDgoYtuC16EzQgVpEMLt6WQu5FI2up3xGVI47KV3OpoRAAMqn8b0STmco3GWw%3D%3D&attredirects=0;https://dl.acm.org/citation.cfm?doid=2391229.2391248)).

[32] *Moxi 1.8 Manual*, Couchbase, Inc., 2014 (<http://docs.couchbase.com/moxi-manual-1.8/>).

[33] Shivnath Babu i Herodotos Herodotou, *Massively Parallel Databases and MapReduce Systems*, „Foundations and Trends in Databases”, rocznik 5, nr 1, s. 1 – 104, listopad 2013 (<https://www.microsoft.com/en-us/research/publication/massively-parallel-databases-and-mapreduce-systems/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F206464%2Fdb-mr-survey-final.pdf>; <http://www.nowpublishers.com/article/Details/DBS-036>).







# Transakcje

*Niektórzy autorzy twierdzą, że obsługa ogólnego dwuetapowego zatwierdzania jest zbyt kosztowna, ponieważ powoduje problemy z wydajnością lub dostępnością. Uważamy, że lepiej jest, by programiści aplikacji rozwiązywali wynikające z nadużywania transakcji problemy z wydajnością, gdy pojawią się wąskie gardła, niż zawsze musieli sobie radzić z brakiem transakcji.*

— James Corbett i in., *Spanner: Google's Globally-Distributed Database* (2012)

W trudnych realiach systemów danych wiele rzeczy może się nie powieść:

- Oprogramowanie bazodanowe lub sprzęt mogą zawieść w każdym momencie (w tym: w trakcie zapisu).
- Aplikacja może w każdej chwili ulec awarii (także w czasie wykonywania serii operacji).
- Zakłócenia w sieci mogą nieoczekiwanie odciąć aplikację od bazy lub jeden węzeł bazy od innego.
- Kilka klientów może zapisywać dane w bazie w tym samym czasie, nadpisując wprowadzane zmiany.
- Klient może wczytać dane, które nie mają sensu, ponieważ zostały tylko częściowo zaktualizowane.
- Sytuacja wyścigu między klientami może powodować zaskakujące błędy.

Aby zapewnić niezawodność, system musi radzić sobie z takimi błędami i gwarantować, że nie wywołają one katastrofalnej awarii całego systemu. Jednak implementowanie mechanizmów zapewniających odporność na błędy wymaga dużo pracy. Konieczne jest staranne przemyślenie wszystkich rzeczy, które mogą się nie udać. Potrzebne są też długie testy gwarantujące, że rozwiązywanie rzeczywiście działa.

Przez dziesięciolecia stosowanym z wyboru mechanizmem do upraszczania takich zagadnień były *transakcje*. Transakcja to sposób grupowania przez aplikację zestawu odczytów i zapisów w jednostkę logiczną. Teoretycznie wszystkie odczyty i zapisy z transakcji są wykonywane jako jedna operacja. Cała transakcja albo kończy się powodzeniem (jest *zatwierdzana*), albo prowadzi do porażki (jest *anulowana* lub *wycofywana*). W przypadku porażki aplikacja może bezpiecznie ponowić próbę. Dzięki transakcjom obsługa błędów w aplikacji staje się znacznie łatwiejsza, ponieważ nie trzeba się obawiać częściowych niepowodzeń, czyli sytuacji, gdy niektóre operacje kończą się sukcesem, a inne porażką (z jakichkolwiek powodów).

Jeśli przez wiele lat posługiwałeś się transakcjami, mogą Ci się wydawać czymś oczywistym. Nie należy jednak brać ich za pewnik. Transakcje nie są prawem natury. Zostały utworzone w określonym celu — aby *uproszczyć model programowania* aplikacji korzystających z baz. Dzięki transakcjom można w aplikacji zignorować niektóre błędy, a także problemy ze współbieżnością, ponieważ za te kwestie odpowiada baza (dająca tzw. *gwarancje bezpieczeństwa*).

Nie każda aplikacja wymaga transakcji. Czasem korzystne jest rozluźnienie gwarancji transakcyjnych lub całkowita rezygnacja z nich (np. w celu poprawy wydajności lub dostępności). Niektóre aspekty bezpieczeństwa można uzyskać bez stosowania transakcji.

Jak ustalić, czy potrzebujesz transakcji? Aby odpowiedzieć na to pytanie, najpierw trzeba dobrze zrozumieć, jakie gwarancje bezpieczeństwa mogą zapewniać transakcje i jakie są koszty stosowania transakcji. Choć początkowo transakcje mogą wydawać się proste, wiążą się z wieloma subtelnymi, ale ważnymi szczegółami.

W tym rozdziale opisano wiele przykładowych scenariuszy, w jakich mogą wystąpić problemy. Omówione są też algorytmy stosowane w bazach w celu ochrony przed takimi problemami. Szczegółowo przedstawiono zwłaszcza obszar kontroli współbieżności. W jego ramach omówiono różne rodzaje sytuacji wyścigu, które mogą wystąpić. Wyjaśniono też, jak w bazach implementowane są poziomy izolacji takie jak *odczyt zatwierdzonych danych*, *izolacja snapshotów* i *sekwencyjność*.

Ten rozdział dotyczy zarówno baz działających w jednym węźle, jak i baz rozproszonych. W rozdziale 8. pokazano konkretne problemy występujące tylko w systemach rozproszonych.

## Niejasne pojęcie transakcji

Obecnie prawie wszystkie bazy relacyjne (a także niektóre nierelacyjne) obsługują transakcje. Większość baz jest zgodna z modelem wprowadzonym w 1975 r. w bazie System R firmy IBM (była to pierwsza baza SQL-owa) [1, 2, 3]. Choć niektóre szczegóły implementacyjne się zmieniły, ogólny pomysł pozostaje prawie taki sam od 40 lat. Obsługa transakcji w bazach MySQL, PostgreSQL, Oracle, SQL Server itd. jest zadziwiająco podobna do rozwiązania z bazy System R.

Pod koniec pierwszej dekady naszego wieku popularność zaczęły zdobywać bazy nierelacyjne (typu NoSQL). Miały być one usprawnieniem w stosunku do standardowych rozwiązań relacyjnych i oferować zestaw nowych modeli danych (zob. rozdział 2.) oraz domyślnie udostępniać replikację (rozdział 5.) i podział na partycje (rozdział 6.). Główną „ofiara” tego ruchu były transakcje. W wielu bazach nowej generacji całkowicie z nich zrezygnowano lub zmieniono definicję tego określenia, łącząc je ze znacznie słabszym niż wcześniej zestawem gwarancji [4].

Wraz z modą na rozproszone bazy danych nowego rodzaju pojawiło się przekonanie, że transakcje są zaprzeczeniem skalowalności i że w każdym dużym systemie trzeba z nich zrezygnować, aby uzyskać wysoką wydajność i dostępność [5, 6]. Z drugiej strony gwarancje transakcyjne są czasem przedstawiane przez producentów baz jako niezbędny wymóg do tworzenia „poważnych aplikacji” z „cennymi danymi”. Obie te opinie są przesadzone.

Prawda nie jest tak prosta. Transakcje, podobnie jak każdy inny wybór w obszarze projektów technicznych, mają zalety i ograniczenia. Aby je zrozumieć, warto się szczegółowo przyjrzeć gwarancjom, jakie transakcje mogą zapewniać — zarówno w normalnym działaniu, jak i różnych skrajnych (ale realistycznych) warunkach.

## Znaczenie gwarancji ACID

Gwarancje bezpieczeństwa zapewniane dzięki transakcjom często opisuje się za pomocą znanego akronimu **ACID** (ang. *atomicity, consistency, isolation, durability*, czyli atomowość, spójność, izolacja i trwałość). Ten akronim został wymyślony w 1983 r. przez Theo Härdera i Andreasa Reutera [7] w celu opracowania precyzyjnej terminologii odnośnie do mechanizmów zapewniania odporności na błędy w bazach.

Jednak w praktyce implementacja gwarancji ACID w jednej bazie nie równa się podobnej implementacji z innej bazy. Na przykład, o czym się przekonasz, określenie *izolacja* jest bardzo wieloznaczne [8]. Ogólny pomysł jest dobry, jednak szczegóły stanowią problem. Obecnie gdy system jest nazywany zgodnym z gwarancjami ACID, nie jest jasne, jakich gwarancji można oczekiwać. ACID stał się, niestety, głównie nazwą marketingową.

Systemy niespełniające kryteriów ACID są czasem nazywane **BASE** (ang. *basically available, soft state i eventual consistency*) [9]. To określenie jest jeszcze bardziej wieloznaczne niż akronim ACID. Wydaje się, że jedyną sensowną definicją BASE jest „nie ACID”. BASE może więc oznaczać prawie cokolwiek.

Przyjrzyjmy się teraz definicjom atomowości, spójności, izolacji i trwałości, ponieważ pozwoli to precyzyjniej zrozumieć transakcje.

### Atomowość

*Atomowość* ogólnie określa coś, czego nie można rozbić na mniejsze części. To słowo oznacza coś podobnego, ale nieco odmiennego w różnych dziedzinach informatyki. Na przykład w programowaniu wielowątkowym jeśli jeden wątek wykonuje operację atomową, nie ma możliwości, aby inny wątek zobaczył wynik w połowie ukończonej operacji. System może znajdować się tylko w stanie sprzed operacji i po niej. Stan pośredni nie występuje.

Natomiast w kontekście gwarancji ACID atomowość *nie* jest związana ze współbieżnością. Nie opisuje, co się dzieje, gdy kilka procesów próbuje uzyskać dostęp do tych samych danych w tym samym czasie, ponieważ tego dotyczy litera *I*, czyli *izolacja* (zob. punkt „Izolacja”).

Atomowość w gwarancjach ACID opisuje, co się dzieje, gdy klient chce wprowadzić kilka zapisów, ale po przetworzeniu niektórych z nich następuje błąd — np. awaria procesu, zakłócenie działania połączenia sieciowego, przepełnienie dysku lub naruszenie więzów integralności. Jeśli zapisy są połączone w atomową transakcję i z powodu błędu nie może ona zostać zakończona (*zatwierdzona*), następuje jej *anulowanie* i baza musi odrzucić lub wycofać wszystkie zapisy wprowadzone w transakcji do danego momentu.

Jeśli atomowość nie jest zapewniona, to po wystąpieniu błędu po wprowadzeniu kilku zmian trudno jest ustalić, które zmiany zostały wprowadzone, a które nie. Aplikacja może ponowić próbę, jednak grozi to dwukrotnym wprowadzeniem tej samej zmiany i powstaniem duplikatów lub błędnych danych. Atomowość upraszcza zadanie. Jeśli transakcja została anulowana, aplikacja ma pewność, że nic się nie zmieniło, dlatego można bezpiecznie ponowić próbę.

Możliwość anulowania transakcji po wystąpieniu błędu i odrzucenia wszystkich zapisów z tej transakcji to definicyjna cecha atomowości z gwarancji ACID. Możliwe, że lepszym od *atomowości* określeniem byłaby *anulowalność*, jednak tu będziemy trzymać się *atomowości*, ponieważ jest to zwykłe słowo.

## Spójność

Słowo *spójność* ma bardzo wiele znaczeń:

- W rozdziale 5. opisano *spójność replik* i kwestię *spójności ostatecznej* występującej w systemach z asynchroniczną replikacją (zob. punkt „Problemy z opóźnieniem replikacji”).
- *Spójne haszowanie* to sposób podziału na partycje stosowany w niektórych systemach do równoważenia (zob. punkt „Spójne haszowanie”).
- W twierdzeniu CAP (zob. rozdział 9.) słowo *spójność* oznacza liniowość (zob. punkt „Liniowość”).
- W kontekście gwarancji ACID słowo *spójność* oznacza specyficzną dla aplikacji ocenę, że baza jest w „dobrym stanie”.

Niedogodne jest to, że jedno słowo ma przynajmniej cztery różne znaczenia.

Idea spójności w gwarancjach ACID jest taka, że występują pewne stwierdzenia na temat danych (*niezmienniki*), które zawsze muszą być prawdziwe. Na przykład w systemie rachunkowości salda kredytowe i debetowe dla wszystkich kont zawsze muszą się równoważyć. Jeśli transakcja zaczyna pracę w bazie, która zgodnie z tymi niezmiennikami jest prawidłowa, a wszystkie zapisy w transakcji zachowują tę poprawność, można mieć pewność, że niezmienniki zawsze będą spełnione.

Jednak taka spójność zależy od tego, czym są niezmienniki w danej aplikacji. Poprawne zdefiniowanie transakcji, tak by zachowywała spójność, należy do obowiązków twórcy aplikacji. Nie jest to coś, co może zagwarantować aplikacja. Jeśli zapiszesz błędne dane naruszające niezmienniki z aplikacji, baza Cię przed tym nie powstrzyma. Niektóre specyficzne rodzaje niezmienników mogą być sprawdzane przez bazę — np. za pomocą więzów klucza obcego lub więzów unikatowości. Jednak ogólnie to aplikacja definiuje, które dane są prawidłowe, a które nie. Baza tylko je przechowuje.

Atomowość, izolacja i trwałość to cechy bazy, natomiast spójność (w znaczeniu stosowanym w kontekście gwarancji ACID) to właściwość aplikacji. Aplikacja może polegać na cechach atomowości i izolacji z bazy danych, aby osiągnąć spójność. Jednak to nie sama baza ma ją zapewniać. Dlatego litera C tak naprawdę nie powinna należeć do akronimu ACID<sup>1</sup>.

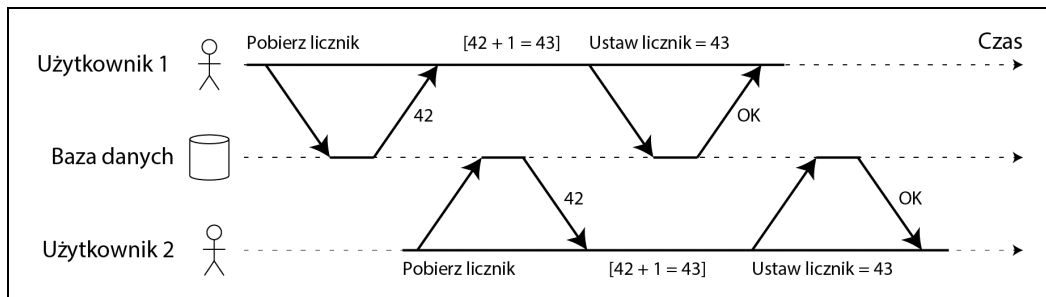
## Izolacja

Większość baz jest używana przez kilka klientów jednocześnie. Nie stanowi to problemu, gdy klienci wczytują i zapisują różne fragmenty bazy. Jeśli jednak używają tych samych rekordów, mogą wystąpić problemy ze współbieżnością (sytuacja wyścigu).

---

<sup>1</sup> Joe Hellerstein stwierdził, że litera „C” w akronimie ACID została „dodana, aby akronim miał sens” w pracy Härdera i Reutera [7] i że wówczas spójność nie była uznawana za istotną.

Rysunek 7.1 przedstawia prosty przykładowy problem tego rodzaju. Załóżmy, że dwa klienci jednocześnie zwiększają wartość przechowywanego w bazie licznika. Każdy klient musi wczytywać aktualną wartość, dodawać 1, a następnie ponownie zapisywać nową wartość (przy założeniu, że baza nie udostępnia wbudowanej operacji inkrementacji). Na rysunku 7.1 wartość licznika powinna wzrosnąć z 42 do 44, ponieważ wykonywane są dwie inkrementacje. Jednak z powodu sytuacji wyścigu wartość rośnie tylko do 43.



Rysunek 7.1. Sytuacja wyścigu między dwoma klientami jednocześnie zwiększającymi wartość licznika

Izolacja w gwarancji ACID oznacza, że jednocześnie wykonywane transakcje są od siebie odizolowane — nie mogą sobie przeszkadzać. W klasycznych podręcznikach z dziedziny baz danych izolacja jest formalnie ujmowana jako *sekwencyjność*, która oznacza, że każda transakcja może udawać, iż jest jedyną transakcją działającą w całej bazie. Baza gwarantuje, że po zatwierdzeniu transakcji efekt będzie taki sam, jak po ich *sekwencyjnym* wykonaniu (jedna po drugiej), choć w rzeczywistości mogą one działać jednocześnie [10].

Jednak w praktyce izolacja w formie sekwencyjności jest rzadko stosowana, ponieważ powoduje spadek wydajności. W niektórych popularnych bazach, np. w Oracle 11g, nie jest ona nawet implementowana. W Oracle występuje poziom izolacji określany jako „sekwencyjność”, ale jednak ma on postać *izolacji snapshotów*, co jest słabszą gwarancją niż sekwencyjność [8, 11]. Izolacja snapshotów i inne formy izolacji są omówione w punkcie „Niższe poziomy izolacji”.

## Trwałość

System bazodanowy ma zapewniać bezpieczne miejsce przechowywania danych bez obaw o ich utratę. *Trwałość* związana jest z obietnicą, że po udanym zatwierdzeniu transakcji zapisane przez nią dane nie zostaną zapomniane — i to nawet po awarii sprzętu lub bazy.

W bazie działającej w jednym węźle zapewnianie trwałości oznacza zwykle, że dane zostały zapisane w trwałej pamięci takiej jak dysk twardy lub dysk SSD. Zwykle stosowany jest też dziennik WAL lub podobny (zob. punkt „Zapewnianie niezawodności b-drzew”) umożliwiający odzyskanie stanu w sytuacji, gdy struktury danych na dysku są uszkodzone. W bazie z replikacją trwałość może oznaczać, że dane zostały z powodzeniem skopiowane do grupy węzłów. Aby zapewnić gwarancje trwałości, baza przed zgłoszeniem transakcji jako udanie zatwierdzonej musi odczekać na zakończenie zapisów lub replikacji.

W punkcie „Niezwadność” opisano, że doskonała trwałość nie jest możliwa. Jeśli wszystkie dyski twarde i kopie zapasowe zostaną zniszczone w tym samym czasie, baza oczywiście nie będzie mogła nic zrobić, aby Cię uratować.

## Replikacja i trwałość

W przeszłości zapewnianie trwałości oznaczało zapis na taśmie archiwalnej. Później zaczęto zapisywać dane na dyskach twardych lub SSD. Od niedawna trwałość wiązana jest z replikacją. Która implementacja jest lepsza?

Prawda jest taka, że nic nie jest doskonałe:

- Jeśli zapisujesz dane na dysku i maszyna przestanie działać, to nawet jeśli dane nie zostaną utracone, będą niedostępne do czasu naprawienia maszyny lub przeniesienia dysku do innego komputera. Systemy z replikacją mogą pozostać dostępne.
- Skorelowana awaria (brak zasilania lub błąd powodujący problemy w każdym węźle po pobraniu określonych danych wejściowych) może jednocześnie wyłączyć wszystkie repliki (zob. punkt „Niezawodność”), co skutkuje utratą wszystkich danych znajdujących się tylko w pamięci. Dlatego zapis na dysku jest ważny nawet przy stosowaniu baz przechowywanych w pamięci.
- W systemach z replikacją asynchroniczną niedawne zapisy mogą zostać utracone, gdy lider stanie się niedostępny (zob. punkt „Radzenie sobie z przestojami węzłów”).
- Po nagłym odcięciu zasilania dyski (zwłaszcza SSD) czasem nie spełniają gwarancji, które powinny zapewniać. Nie ma nawet pewności, że poprawnie zadziała wywołanie `fsync` [12]. W oprogramowaniu dysku mogą występować błędy takie same jak w każdym innym kodzie [13, 14].
- Subtelne interakcje między systemem składowania danych a implementacją systemu plików mogą prowadzić do trudnych do wykrycia błędów. Mogą też spowodować, że po awarii pliki na dysku będą uszkodzone [15, 16].
- Dane na dysku mogą być stopniowo uszkodzane bez wykrycia tego faktu [17]. Jeśli dane są uszkodzone przez pewien czas, repliki i nowe kopie zapasowe też mogą być nieprawidłowe. Wtedy konieczna jest próba odzyskania danych z historycznej kopii zapasowej.
- W jednym z badań nad dyskami SSD stwierdzono, że w przypadku 30% – 80% takich urządzeń w ciągu czterech pierwszych lat pracy pojawia się przynajmniej jeden błędny blok [18]. W magnetycznych dyskach twardych odsetek błędnych sektorów jest niższy, jednak całkowite awarie zdarzają się częściej niż w dyskach SSD.
- Gdy dysk SSD jest odłączony od zasilania, po kilku tygodniach może zacząć tracić dane (zależy to od temperatury) [19].

W praktyce żadna technika nie zapewnia absolutnych gwarancji. Istnieją tylko różne metody ograniczenia ryzyka, w tym zapis na dysku, replikacja z użyciem zdalnych maszyn i kopie zapasowe. Wszystkie te techniki można i należy stosować razem. Jak zawsze warto traktować wszelkie gwarancje „teoretyczne” ze zdrową rezerwą.

## Operacje na pojedynczych obiektach i na wielu obiektach

W podsumowaniu można stwierdzić, że w modelu ACID atomowość i izolacja określają, co baza powinna robić, jeśli klient wykonuje kilka zapisów w tej samej transakcji:

### Atomowość

Jeśli w trakcie wykonywania sekwencji zapisów wystąpi błąd, transakcję należy anulować i odrzucić zapisy wprowadzone do danego momentu. Oznacza to, że baza oferuje gwarancje „wszystko lub nic” i chroni w ten sposób przed koniecznością martwienia się o częściowe niepowodzenie.

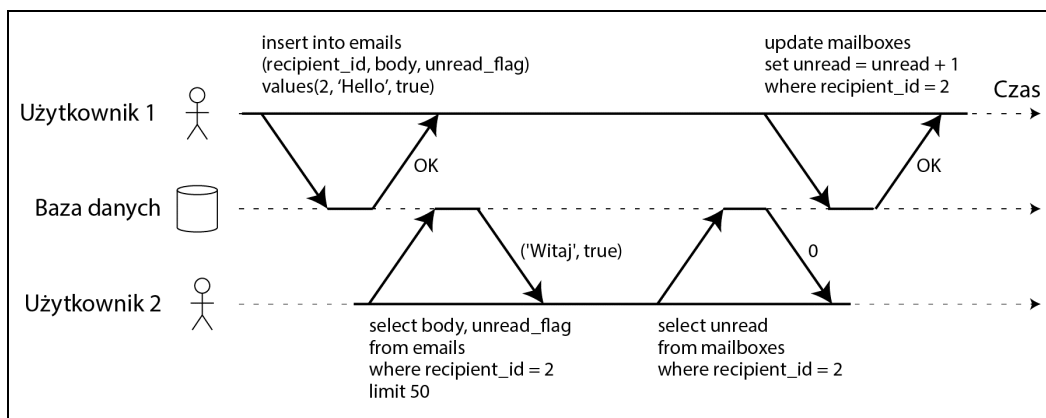


## Izolacja

Jednocześnie wykonywane transakcje nie powinny zakłócać swojej pracy. Na przykład jeśli jedna transakcja wykonuje kilka zapisów, inna albo powinna zobaczyć je wszystkie, albo nie powinna widzieć żadnego z nich. Nie może zobaczyć tylko podzbioru tych zapisów.

W tych definicjach przyjęto, że cel stanowi jednoczesne zmodyfikowanie kilku obiektów (wierszy, dokumentów, rekordów). Tego typu *transakcje na wielu obiektach* często są potrzebne, gdy konieczna jest synchronizacja różnych danych. Na rysunku 7.2 pokazano przykład działania klienta poczty elektronicznej. Aby wyświetlić liczbę nieprzeczytanych wiadomości do użytkownika, można wywołać następujące zapytanie:

```
SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
```



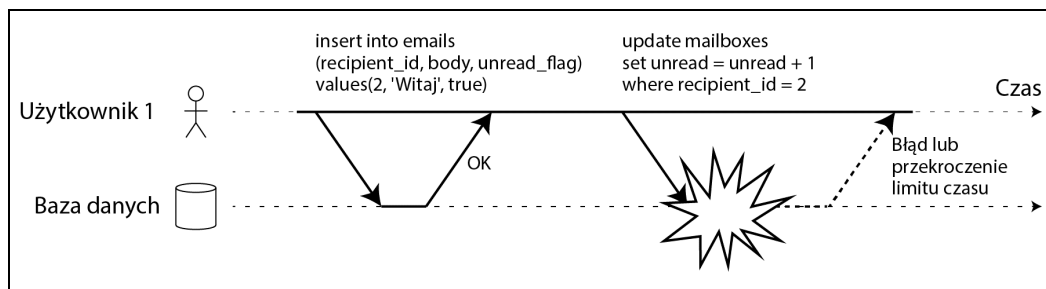
Rysunek 7.2. Naruszenie izolacji — jedna transakcja wczytuje niezatwierdzone zapisy innej transakcji („brudny odczyt”)

Jeśli jednak liczba e-maili jest duża, zapytanie może być zbyt powolne. Możesz się wtedy zdecydować zapisać liczbę nieprzeczytanych wiadomości w odrębnym polu (jest to rodzaj denormalizacji). Teraz gdy pojawi się nowa wiadomość, trzeba zwiększyć wartość licznika nieprzeczytanych e-maili, a każde oznaczenie wiadomości jako przeczytanej wymaga zmniejszenia wartości tego licznika.

W sytuacji z rysunku 7.2 użytkownik 2 widzi anomalię. W skrzynce pocztowej widoczna jest nieprzeczytana wiadomość, jednak licznik pokazuje zero nieprzeczytanych e-maili, ponieważ jego wartość nie została jeszcze zwiększona<sup>2</sup>. Izolacja zapobiegłaby temu problemowi, gwarantując, że użytkownik albo zobaczy przesłany e-mail i zaktualizowany licznik, albo nie zobaczy żadnej z tych rzeczy. Niespójny połowiczny obraz nie jest możliwy.

Na rysunku 7.3 pokazano, dlaczego potrzebna jest atomowość. Jeśli błąd nastąpi w czasie wykonywania transakcji, zawartość skrzynki pocztowej i licznik nieprzeczytanych wiadomości mogą się stać niesynchronizowane. W transakcji atomowej niepowodzenie aktualizacji licznika skutkuje anulowaniem transakcji i wycofaniem dodanego e-maila.

<sup>2</sup> Można stwierdzić, że nieprawidłowa wartość licznika w kliencie poczty elektronicznej nie jest wysoce krytycznym problemem. Możesz więc zastąpić licznik stanem konta klienta, a e-mail transakcją płatniczą.



Rysunek 7.3. Atomowość gwarantuje, że jeśli wystąpi błąd, wszystkie wcześniejsze zapisy z danej transakcji zostaną wycofane, aby uniknąć niespójnego stanu

Transakcje na wielu obiektach wymagają sposobu na określenie, które operacje odczytu i zapisu należą do tej samej transakcji. W bazach relacyjnych zwykle odbywa się to na podstawie połączenia TCP między klientem a serwerem bazodanowym. Dla danego połączenia wszystko między instrukcjami `BEGIN TRANSACTION` a `COMMIT` jest uznawane za część tej samej transakcji<sup>3</sup>.

Z kolei w wielu bazach nierelacyjnych nie występuje taka metoda grupowania operacji. Nawet gdy istnieje interfejs API dla wielu obiektów (np. baza z parami klucz-wartość może udostępniać polecenie *zapisu wielu elementów*, które aktualizuje kilka kluczy w jednej operacji), nie musi to oznaczać, że działa on tak jak transakcje. Polecenie może się zakończyć powodzeniem dla niektórych kluczy i niepowodzeniem dla innych, przez co baza będzie częściowo zaktualizowana.

## Zapisy pojedynczego obiektu

Atomowość i izolacja mają też zastosowanie przy modyfikowaniu pojedynczego obiektu. Wyobraź sobie np., że zapisujesz w bazie dokument w formacie JSON o wielkości 20 KB:

- Jeśli połączenie sieciowe zostanie zerwane po przesłaniu pierwszych 10 KB, czy baza zapisze te niemożliwe do przetworzenia 10 KB w formacie JSON?
- Jeżeli nastąpi utrata zasilania w trakcie nadpisywania przez bazę wcześniejszej wartości na dysku, czy dawna i nowa wartość zostaną scalone?
- Jeśli inny klient wczyta dokument w trakcie zapisu, czy zobaczy częściowo zaktualizowaną wartość?

Te sytuacje byłyby bardzo kłopotliwe, dlatego systemy składowania danych prawie zawsze próbują zapewniać atomowość i izolację na poziomie pojedynczego obiektu (np. pary klucz-wartość) w jednym węźle. Atomowość można zaimplementować, używając dziennika do odzyskiwania stanu po awarii (zob. punkt „Zapewnianie niezawodności b-drzew”). Izolację można zapewnić, stosując blokadę dla każdego obiektu (pozwalać w danym momencie tylko jednemu wątkowi na dostęp do obiektu).

<sup>3</sup> Nie jest to idealne rozwiązanie. Jeśli połączenie TCP zostanie przerwane, transakcję trzeba anulować. Jeżeli zerwanie połączenia nastąpiło po tym, jak klient zażądał zatwierdzenia, ale przed potwierdzeniem zatwierdzenia przez serwer, klient nie wie, czy dana transakcja została zatwierdzona. Aby rozwiązać ten problem, menedżer transakcji może grupować operacje według unikatowego identyfikatora, który nie jest powiązany z konkretnym połączeniem TCP. Do tego tematu wrócimy w punkcie „Zasada punktów końcowych dla baz danych”.

Niektóre bazy obsługują też bardziej złożone operacje atomowe<sup>4</sup>, np. inkrementację, dzięki czemu nie jest konieczny przedstawiony na rysunku 7.1 cykl „wczytaj – zmodyfikuj – zapisz”. Popularna jest też operacja „porównaj i ustaw”, która dopuszcza zapis tylko, jeśli wartość nie została jednocześnie zmodyfikowana przez inną jednostkę (zob. punkt „Porównaj i ustaw”).

Takie operacje na pojedynczych obiektach są przydatne, ponieważ mogą zapobiec utracie aktualizacji, gdy kilka klientów próbuje jednoczesnego zapisu tego samego obiektu (zob. punkt „Zapobieganie utracie aktualizacji”). Nie są to jednak transakcje w tradycyjnym tego słowa znaczeniu. „Porównaj i ustaw” oraz inne operacje na pojedynczych obiektach są na potrzeby marketingu nazywane „lekkimi transakcjami” lub nawet „ACID” [20, 21, 22]. Jednak ta terminologia jest myląca. Transakcje są zwykle rozumiane jako mechanizm grupowania zestawu operacji na wielu obiektach w odrębną jednostkę wykonawczą.

### Potrzeba stosowania transakcji dla wielu obiektów

W licznych bazach i grupach obiektów zrezygnowano z transakcji na wielu obiektach, ponieważ trudno jest je zaimplementować z uwzględnieniem wielu partycji. Ponadto mogą stwarzać problemy w niektórych scenariuszach, gdy potrzebna jest bardzo wysoka dostępność lub wydajność. Jednak zasadniczo nic nie stoi na przeszkodzie, by stosować transakcje w bazach rozproszonych. Implementacje transakcji rozproszonych są opisane w rozdziale 9.

Jednak czy w ogóle potrzebne są transakcje na wielu obiektach? Czy możliwe byłoby zaimplementowanie dowolnej aplikacji wyłącznie za pomocą modelu danych i operacji na pojedynczych obiektach?

Istnieją pewne przypadki użycia, w których wstawianie, aktualizowanie i usuwanie pojedynczych obiektów jest wystarczające. Jednak w licznych innych scenariuszach trzeba koordynować zapis wielu różnych obiektów:

- W relacyjnym modelu danych wiersz z jednej tabeli często zawiera referencję w postaci klucza obcego prowadzącego do wiersza z innej tabeli (podobnie w modelu danych opartym na grafach wierzchołek jest połączony krawędziami z innymi wierzchołkami). Transakcje na wielu obiektach pozwalają zagwarantować, że te referencje pozostaną poprawne. Gdy wstawiasz kilka rekordów powiązanych kluczami obcymi, te klucze muszą być poprawne i aktualne. W przeciwnym razie dane staną się bezsensowne.
- W modelu danych opartym na dokumentach pola, które trzeba aktualizować wspólnie, pola często znajdują się w tym samym dokumencie. Ten dokument jest traktowany jak jeden obiekt. W trakcie aktualizowania pojedynczego dokumentu nie są potrzebne transakcje na wielu obiektach. Jednak bazy oparte na dokumentach, które nie obejmują mechanizmu złączania, skłaniają do denormalizacji (zob. punkt „Bazy relacyjne a bazy oparte na dokumentach obecnie”). Gdy konieczna jest aktualizacja zdenormalizowanych informacji (tak jak na rysunku 7.2, trzeba zaktualizować kilka dokumentów w jednym kroku. Transakcje są wtedy bardzo przydatne, aby zapobiec desynchronizacji zdenormalizowanych danych.

---

<sup>4</sup> Ścisłe rzecz biorąc, w określeniu *atomowa inkrementacja* słowo *atomowa* jest używane w sensie typowym dla programowania wielowątkowego. W kontekście gwarancji ACID powinno się mówić o inkrementacji *izolowanej* lub *sekwencyjnej*. Jednak jest to już czepianie się.

- W bazach z indeksami pomocniczymi (czyli w prawie wszystkich oprócz czystych baz z parami klucz-wartość) każda zmiana wartości wymaga też zaktualizowania indeksów. Z perspektywy transakcji te indeksy są obiektami bazy. Bez izolowania transakcji może się zdarzyć, że rekord pojawi się w jednym indeksie, ale już nie w drugim, ponieważ ten drugi nie został jeszcze zaktualizowany.

Aplikacje tego rodzaju też można zaimplementować bez transakcji. Jednak bez atomowości obsługa błędów staje się dużo trudniejsza, a brak izolacji może skutkować problemami ze współbieżnością. Te zagadnienia są opisane w punkcie „Niższe poziomy izolacji”, a omówienie innych rozwiązań zawiera rozdział 12.

## Obsługa błędów i anulowania

Ważną cechą transakcji jest to, że po wystąpieniu błędu można ją anulować i bezpiecznie ponowić. Na tym podejściu są oparte bazy zgodne z modelem ACID. Jeśli bazie grozi naruszenie gwarancji atomowości, izolacji lub trwałości, należy całkowicie zrezygnować z transakcji, zamiast pozostawiać ją ukończoną w połowie.

Jednak nie wszystkie systemy działają zgodnie z tym podejściem. Zwłaszcza bazy z replikacją bez lidera (zob. punkt „Replikacja bez lidera”) znacznie częściej funkcjonują na zasadzie „dostępnych możliwości”. Można ją podsumować tak: baza robi wszystko, co możliwe, a jeśli wystąpi błąd, nie wycofa operacji, które już zostały wykonane. Dlatego to aplikacja odpowiada za odzyskanie stanu po błędach.

Błędy bez wątpienia będą się zdarzać, jednak wielu programistów aplikacji woli myśleć tylko o ścieżce optymistycznej zamiast o zawiłościach obsługi błędów. Na przykład popularne platformy **ORM** (ang. *object-relational mapping*), takie jak ActiveRecord dla środowiska Rails lub Django, nie ponawiają anulowanych transakcji. Błąd zwykle prowadzi do przekazywania wyjątku w górę stosu, dlatego dane wejściowe są odrzucane i użytkownik otrzymuje komunikat o błędzie. Jest to niekorzystne, ponieważ anulowanie ma właśnie umożliwiać bezpieczne ponawianie prób.

Choć ponawianie anulowanej transakcji to prosty i skuteczny mechanizm obsługi błędów, nie jest doskonałym rozwiązaniem, ponieważ:

- Jeśli transakcja zakończyła się powodzeniem, ale sieć przestała działać, gdy serwer próbował potwierdzić klientowi udane zatwierdzenie operacji (przez co klient sądzi, że zadanie się nie powiodło), ponowienie transakcji spowoduje jej dwukrotne wykonanie — chyba że na poziomie aplikacji stosowany jest dodatkowy mechanizm unikania powtórzeń.
- Jeśli błąd wynika z przeciążenia, ponowienie transakcji nasila problem, zamiast poprawiać sytuację. Aby uniknąć tego rodzaju cykli, można ograniczyć liczbę ponawianych prób, zastosować odczekiwanie wykładnicze i obsługiwać błędy związane z przeciążeniem inaczej niż odmienne trudności (jeśli jest to możliwe).
- Próbę warto ponowić tylko po przejściowym problemie (np. spowodowanym zakleszczeniem, naruszeniem izolacji, tymczasowymi zakłóceniami w sieci lub przełączaniem awaryjnym). Trwały błąd (np. naruszenie więzów) sprawia, że ponawianie prób jest bezcelowe.
- Jeżeli transakcja powoduje efekty uboczne poza bazą, mogą one wystąpić nawet po anulowaniu transakcji. Jeśli np. przesyłasz e-mail, nie chcesz robić tego przy każdym ponawianiu transakcji.

Gdy chcesz mieć pewność, że kilka różnych systemów wspólnie zatwierdziło lub anulowało operację, pomocne może być zatwierdzanie dwuetapowe (omówione w punkcie „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”).

- Gdy proces klienta zawiedzie w trakcie ponawiania próby, dane, które proces próbował zapisać w bazie, zostaną utracone.

## Niskie poziomy izolacji

Jeśli dwie transakcje nie dotyczą tych samych danych, można je bezpiecznie uruchamiać równolegle, ponieważ żadna z nich nie zależy od drugiej. Problemy ze współbieżnością (sytuacja wyścigu) występują tylko wtedy, gdy jedna transakcja wczytuje dane jednocześnie modyfikowane przez inną lub gdy dwie transakcje próbują w tym samym czasie modyfikować te same dane.

Błędy związane ze współbieżnością są trudne do wykrycia w trakcie testów, ponieważ zachodzą tylko przy niekorzystnego synchronizowania operacji w czasie. Takie problemy z synchronizacją mogą występować bardzo rzadko i zwykle są trudne do odtworzenia. Ponadto analiza kodu współbieżnego jest bardzo uciążliwa — zwłaszcza w dużych aplikacjach, bo w nich nie zawsze wiadomo, które fragmenty kodu korzystają z bazy. Tworzenie aplikacji jest wystarczająco skomplikowane, jeśli użytkownik jest tylko jeden. Wielu równoległych użytkowników znacznie utrudnia zadanie, ponieważ dowolny fragment danych może w dowolnej chwili nieoczekiwanie się zmienić.

Z tego powodu w bazach od dawna starano się ukrywać problemy ze współbieżnością przed programistami aplikacji, zapewniając *izolację transakcji*. Teoretycznie izolacja powinna ułatwiać pracę, ponieważ pozwala udawać, że współbieżność nie zachodzi. Izolacja oparta na *sekwencyjności* oznacza, że baza gwarantuje, iż transakcje dają ten sam efekt jak w wyniku ich *sekwencyjnego* wykonania (czyli jedna po drugiej, bez współbieżności).

Niestety, w praktyce izolacja nie jest taka prosta. Izolacja oparta na sekwencyjności powoduje spadek wydajności i twórcy wielu baz tego nie akceptują [8]. Dlatego w systemach często stosowane są niższe poziomy izolacji chroniące przed *niektórymi* problemami ze współbieżnością, ale nie przed wszystkimi. Te poziomy izolacji znacznie trudniej jest zrozumieć. Ponadto mogą one prowadzić do subtelnych błędów, jednak mimo to stosuje się je w praktyce [23].

Błędy z obszaru współbieżności spowodowane słabą izolacją transakcji nie są problemem wyłącznie teoretycznym. Doprowadziły one do utraty znacznych kwot pieniędzy [24, 25], dociekań audytorów finansowych [26] i uszkodzenia danych klientów [27]. Po ujawnieniu tego rodzaju problemów często można usłyszeć komentarz: „Jeśli zarządzacie danymi finansowymi, stosujcie bazy ACID!”. Jednak nie o to tu chodzi. Nawet w wielu popularnych bazach relacyjnych (zwykle uważanych za zgodne z modelem ACID) stosowana jest słaba izolacja. Dlatego takie bazy niekoniecznie zapobiegłyby powstaniu opisanych błędów.

Zamiast ślepo polegać na narzędziach, należy dobrze zrozumieć rodzaje problemów ze współbieżnością i sposoby zapobiegania im. Następnie można budować niezawodne i prawidłowe aplikacje, posługując się dostępnymi narzędziami.

W tym punkcie omówiono kilka stosowanych w praktyce niskich poziomów izolacji (bez sekwen-  
cyjności). Opisano też szczegółowo, jakiego rodzaju sytuacje wyścigu mogą się przytrafić, a jakie nie  
występują. Dzięki temu będziesz mógł zdecydować, który poziom będzie odpowiedni w Twojej  
aplikacji. Dalej szczegółowo omówiona jest sekwencyjność (zob. punkt „Sekwencyjność”). Opis  
poziomów izolacji jest tu nieformalny i bazuje na przykładach. Jeśli szukasz ścisłych definicji i analiz  
cech, znajdziesz te informacje w podręcznikach akademickich [28, 29, 30].

## Odczyt zatwierdzonych danych

Najbardziej podstawowym poziomem izolowania transakcji jest *odczyt zatwierdzonych danych*  
(ang. *read committed*)<sup>5</sup>. Wiąże się on z dwiema gwarancjami:

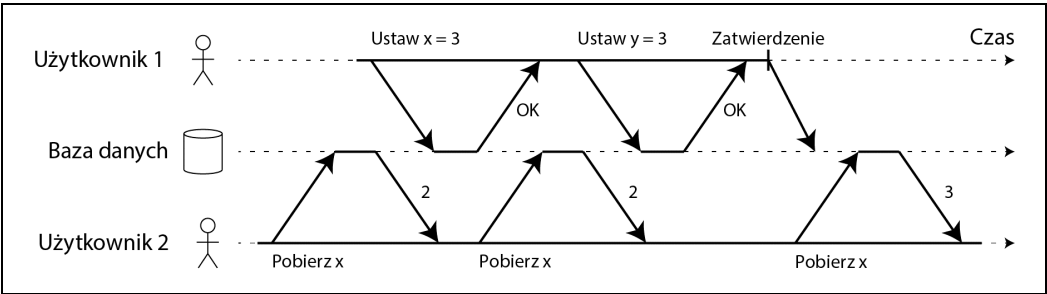
1. W trakcie odczytu z bazy widoczne są tylko zatwierdzone dane (brak *brudnych odczytów*).
2. W czasie zapisu w bazie nadpisywane są tylko zatwierdzone dane (brak *brudnych zapisów*).

Warto szczególnie omówić te dwie gwarancje.

### Brak brudnych odczytów

Wyobraź sobie, że transakcja zapisała jakieś dane w bazie, ale nie została jeszcze zatwierdzona ani  
anulowana. Czy inna transakcja może zobaczyć te niezatwierdzone dane? Jeśli tak, następuje *brudny*  
*odczyt* [2].

Transakcje działające na poziomie izolacji „odczyt zatwierdzonych danych” muszą zapobiegać  
brudnym odczytom. To oznacza, że zapisy dokonywane przez transakcję stają się widoczne dla  
innych jednostek dopiero po zatwierdzeniu tej transakcji (wtedy wszystkie te zapisy stają się do-  
stępne jednocześnie). Ilustruje to rysunek 7.4, gdzie użytkownik 1 ustawił  $x = 3$ , ale dopóki użyt-  
kownik 1 nie zatwierdzi transakcji, operacja *pobierz x* wykonana przez użytkownika 2 nadal zwraca  
dawną wartość (2).



Rysunek 7.4. Brak brudnych odczytów — użytkownik 2 widzi nową wartość  $x$  dopiero po zatwierdzeniu  
transakcji użytkownika 1

Jest kilka powodów, dla których warto zapobiegać brudnemu odczytowi:

- Jeśli transakcja musi zaktualizować kilka obiektów, brudny odczyt oznacza, że inna transakcja może zobaczyć niektóre aktualizacje, ale już nie inne. Na przykład na rysunku 7.2 użytkownik

<sup>5</sup> Niektóre bazy zapewniają jeszcze niższy poziom izolacji, *odczyt niezatwierdzonych danych*. Zapobiega on brudnemu za-  
pisowi, ale nie chroni przed brudnym odczytem.

widzi nowy nieprzeczytany e-mail, ale licznik nie jest zaktualizowany. Jest to brudny odczyt e-maila. Dostęp do częściowo zaktualizowanej bazy jest mylący dla użytkowników i może powodować, że inne transakcje będą podejmować niewłaściwe decyzje.

- Jeżeli transakcja jest anulowana, wszystkie wprowadzone w niej zapisy trzeba wycofać (tak jak na rysunku 7.3). Jeśli baza dopuszcza brudne odczyty, oznacza to, że transakcja może zobaczyć dane, które zostaną później usunięte (czyli takie, które nigdy nie zostały zatwierdzone w bazie). Analizowanie konsekwencji takiej sytuacji może szybko przyprawić o ból głowy.

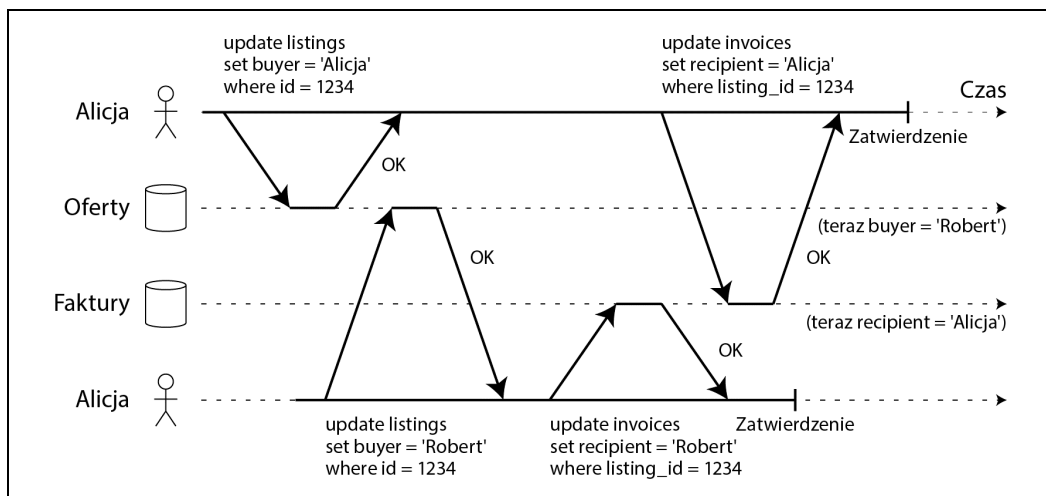
## Brak brudnych zapisów

Co się stanie, jeśli dwie transakcje jednocześnie spróbują zaktualizować ten sam obiekt w bazie? Nie wiadomo, w jakiej kolejności zapisy będą wprowadzane. Jednak standardowo przyjmuje się, że wcześniejszy zapis jest zastępowany przez późniejszy.

Co się jednak dzieje, gdy wcześniejszy zapis jest częścią niezatwierdzonej transakcji, a późniejszy nadpisuje niezatwierdzoną wartość? Jest to tzw. *brudny zapis* [28]. Transakcje działające na poziomie izolacji „odczyt zatwierdzonych danych” muszą zapobiegać brudnym odczytom — zwykle odraczając drugi zapis do momentu zatwierdzenia lub anulowania transakcji z pierwszym zapisem.

Dzięki zapobieganiu brudnym zapisom ten poziom izolacji pozwala uniknąć niektórych rodzajów problemów ze współbieżnością:

- Jeśli transakcja aktualizuje wiele obiektów, brudny zapis może prowadzić do nieprawidłowych skutków. Zastanów się nad rysunkiem 7.5. Pokazana jest tam witryna firmy sprzedającej używane samochody. W tej witrynie dwie osoby, Alicja i Robert, jednocześnie próbują kupić ten sam samochód. Zakup auta wymaga dwóch zapisów w bazie: trzeba zaktualizować ofertę w witrynie, aby uwzględnić kupca, a także przesłać kupcowi fakturę sprzedaży. Na rysunku 7.5 za kupca zostaje uznany Robert (ponieważ przeprowadził „zwycięską” aktualizację tabeli listings z ofertami), ale faktura trafia do Alicji (ponieważ wykonała „zwycięską” aktualizację tabeli invoices z fakturami). Odczyt zatwierdzonych danych chroni przed takimi wpadkami.



Rysunek 7.5. Brudne zapisy mogą spowodować wymieszanie sprzecznych zapisów z różnych transakcji

- Jednak odczyt zatwierdzonych danych *nie* zapobiega sytuacji wyścigu między dwoma inkrementacjami licznika na rysunku 7.1. W tym scenariuszu drugi zapis odbywa się po zatwierdzeniu pierwszej transakcji, nie jest więc brudnym zapisem. Jest nieprawidłowy, jednak z innego powodu. W punkcie „Zapobieganie utracie aktualizacji” wyjaśniono, jak zapewnić bezpieczeństwo takim inkrementacjom licznika.

## Implementowanie odczytu zatwierdzonych danych

Odczyt zatwierdzonych danych to bardzo często stosowany poziom izolacji. Jest to domyślne ustawienie w bazach Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL i wielu innych [8].

Bazy najczęściej zapobiegają brudnym zapisom za pomocą blokad z poziomu wierszy. Gdy transakcja chce zmodyfikować konkretny obiekt (wiersz lub dokument), musi najpierw uzyskać blokadę tego obiektu. Następnie musi utrzymać tę blokadę do momentu zatwierdzenia lub anulowania transakcji. Tylko jedna transakcja może utrzymywać blokadę danego obiektu. Jeśli inna transakcja chce zapisać dane w tym samym obiekcie, musi odczekać do momentu zatwierdzenia lub anulowania pierwszej transakcji. Dopiero wtedy może uzyskać blokadę i kontynuować pracę. W trybie odczytu zatwierdzonych danych (i przy wyższych poziomach izolacji) blokady są obsługiwane automatycznie przez bazy.

Jak zapobiegać brudnym odczytom? Jedną z możliwości jest wykorzystanie tej samej blokady i wymóg, by każda transakcja, która chce wczytać obiekt, na krótko zajmowała blokadę, a następnie natychmiast po odczycie ją zwalniała. To gwarantuje, że odczyt nie nastąpi w czasie, gdy obiekt zawiera „brudną”, niezatwierdzoną wartość (ponieważ wtedy blokada byłaby zajmowana przez transakcję odpowiedzialną za zapis).

Jednak w praktyce podejście wymagające blokad na czas odczytu nie sprawdza się dobrze, ponieważ jedna długa transakcja zapisu może wymagać oczekiwania na jej zakończenie przez wiele transakcji tylko wczytujących dane. Wydłuża to czas przetwarzania transakcji tylko wczytujących dane i jest złe w kontekście eksploatacji. Z powodu oczekiwania na blokady spowolnienie w jednej części aplikacji może wywołać efekt domina w zupełnie innym miejscu.

Dlatego większość baz<sup>6</sup> zapobiega brudnym odczytom za pomocą podejścia przedstawionego na rysunku 7.4. Dla każdego zapisywanego obiektu baza zapamiętuje zarówno dawną zatwierdzoną wartość, jak i nową wartość ustawioną przez transakcję, która aktualnie zajmuje blokadę. W trakcie wykonywania transakcji wszystkie inne transakcje wczytujące obiekt otrzymują dawną wartość. Dopiero po zatwierdzeniu nowej wartości transakcje zaczynają ją wczytywać.

## Izolacja snapshotów i powtarzalny odczyt

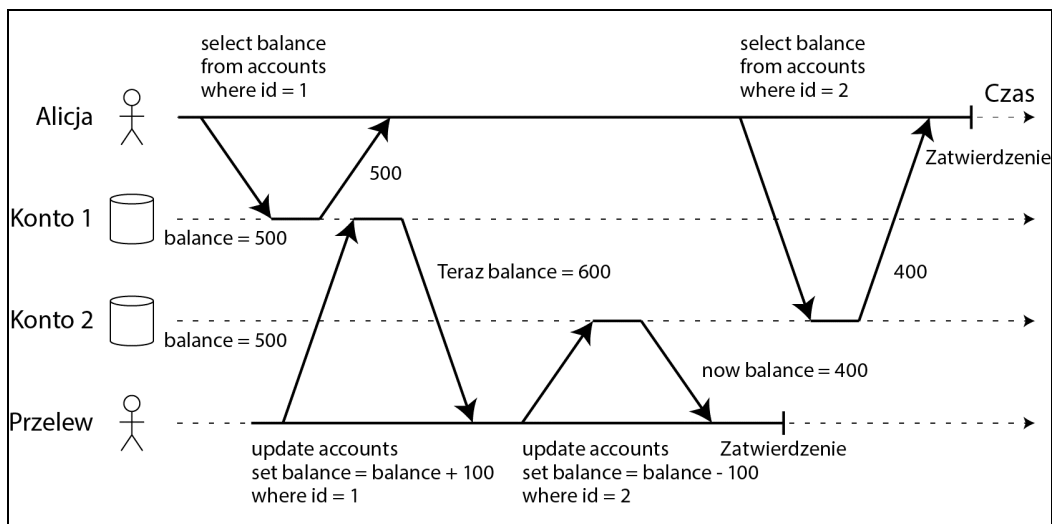
Jeśli tylko pobieżnie zastanowiłeś się nad izolacją w formie odczytów zatwierdzonych wartości, możesz się usprawiedliwić z myślenia, że jest to wszystko, czego aplikacja potrzebuje. Ten poziom umożliwia anulowanie (potrzebne do uzyskania atomowości), zapobiega odczytowi niekompletnych wyników transakcji i chroni przed wymieszaniem jednoczesnych zapisów. Rzeczywiście, są to przydatne cechy, zapewniające znacznie mocniejsze gwarancje, niż można uzyskać w systemie, w którym transakcje nie są używane.

---

<sup>6</sup> W czasie, gdy powstaje ta książka, jedynymi popularnymi bazami używającymi blokad do izolacji typu „odczyt zatwierdzonych danych” są DB2 firmy IBM i SQL Server Microsoftu w konfiguracji `read_committed_snapshot=off` [23, 26].



Mimo to na tym poziomie izolacji i tak występuje bardzo dużo scenariuszy, które mogą prowadzić do błędów współbieżności. Na przykład na rysunku 7.6 przedstawiony jest problem, który może wystąpić na poziomie odczytu zatwierdzonych danych.



Rysunek 7.6. Odczyt zniekształcony — Alicja widzi bazę w niespójnym stanie

Żałujemy, że Alicja przechowuje w banku 1000 zł oszczędności rozbitych między dwa konta po 500 zł. Transakcja przenosi 100 zł z jednego z tych kont na drugie. Jeśli Alicja nieszczęśliwie sprawdzi swój rachunek w momencie przetwarzania transakcji, możliwe, że zobaczy saldo jednego konta z momentu przed dotarciem przelewu (stan 500 zł) i saldo drugiego konta po ściągnięciu z niego środków (z nowym stanem 400 zł). Dla Alicji wygląda to tak, jakby w sumie miała na rachunkach tylko 900 zł. Wydaje się, że 100 zł po prostu zniknęło.

Ta anomalia to *niepowtarzalny odczyt* lub *odczyt zniekształcony*. Jeśli Alicja ponownie wczyta stan konta 1 po zakończeniu transakcji, zobaczy inną wartość (600 zł) niż we wcześniejszym zapytaniu. Na poziomie izolacji „odczyt zatwierdzonych danych” odczyt zniekształcony jest uznawany za akceptowalny. Stan kont, jaki zobaczyła Alicja, rzeczywiście był zatwierdzony w momencie ich odczytu.

Problem Alicji nie jest długotrwały, ponieważ prawdopodobnie zobaczy ona spójne stany kont, jeśli po kilku sekundach odświeży witrynę banku. Jednak w niektórych sytuacjach taka tymczasowa niespójność jest nie do przyjęcia.

#### Kopie zapasowe

Przygotowanie kopii zapasowej wymaga skopiowania całej bazy. Jeśli baza jest duża, może to trwać godzinami. W trakcie pracy procesu tworzącego kopię zapasową dane w bazie wciąż są zapisywane. Dlatego niektóre części kopii zapasowej mogą obejmować starszą wersję danych, a inne — nowszą. Jeżeli będziesz musiał odzyskać stan z użyciem takiej kopii zapasowej, niespójności (np. utrata środków) staną się trwałe.

Czasem chcesz uruchomić zapytanie, które skanuje duże części bazy. Takie zapytania są częste w analityce (zob. punkt „Przetwarzanie transakcji czy analityka?”) lub mogą być częścią okresowego sprawdzania integralności w celu ustalenia, czy wszystko działa poprawnie (system szuka wtedy uszkodzonych danych). Takie zapytania zapewne zwrócą bezsensowne wyniki, jeśli zobaczą części bazy z różnych momentów w czasie.

Najczęstszym rozwiązaniem tego problemu jest *izolacja snapshotów* [28]. Pomysł polega na tym, by każda transakcja czytywała dane ze *spójnego snapshota* bazy. Transakcja widzi więc wszystkie dane zatwierdzone w bazie w momencie rozpoczęcia tej transakcji. Nawet jeśli dane zostaną później zmodyfikowane przez inną transakcję, każda transakcja widzi tylko wcześniejsze dane z określonego punktu w czasie.

Izolacja snapshotów to prawdziwe dobrodziejstwo w przypadku długich zapytań w trybie tylko do odczytu (służących np. do analityki i tworzenia kopii zapasowych). Bardzo trudno jest analizować znaczenie zapytania, jeśli dane, które przetwarza, zmieniają się w trakcie wykonywania go. Gdy transakcja może zobaczyć zamrożony spójny snapshot bazy z określonego momentu, znacznie łatwiej jest zrozumieć zapytanie.

Izolacja snapshotów to popularna funkcja obsługiwana w bazach PostgreSQL, Oracle, SQL Server, MySQL z systemem składowania danych InnoDB i in. [23, 31, 32].

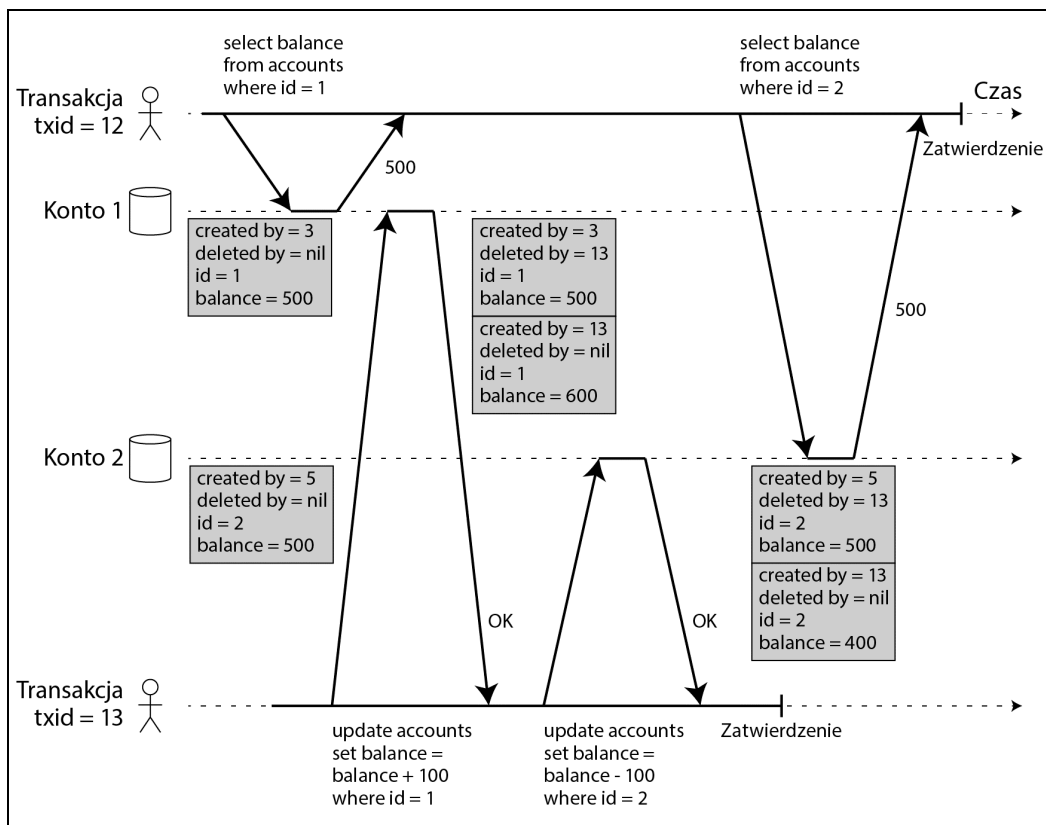
### Implementowanie izolacji snapshotów

W implementowaniu izolacji snapshotów (podobnie jak odczytu zatwierdzonych danych) wykorzystuje się zwykle blokady zapisu, aby zapobiegać brudnym zapisom (zob. punkt „Implementowanie odczytu zatwierdzonych danych”). Oznacza to, że transakcja zapisująca dane może zablokować pracę innej transakcji zapisującej ten sam obiekt. Jednak odczyty nie wymagają żadnych blokad. W kontekście wydajności podstawową zasadą izolacji snapshotów jest to, że *jednostki odczytujące nigdy nie blokują jednostek zapisujących i na odwrót*. Dzięki temu baza może wykonywać długie zapytania z odczytem, posługując się spójnym snapshotem, a w tym samym czasie normalnie przetwarzać zapisy bez współzawodniczenia o blokadę między tymi operacjami.

Aby zaimplementować izolację snapshotów, w bazach stosowane jest uogólnienie pokazanego na rysunku 7.4 mechanizmu zapobiegania brudnym odczytom. Baza potencjalnie musi przechowywać kilka zatwierdzonych wersji obiektu, ponieważ różne trwające transakcje muszą widzieć stan bazy z innego czasu. Ponieważ baza przechowuje równolegle kilka wersji obiektu, ta technika jest nazywana MVCC (ang. *multiversion concurrency control*, czyli zarządzanie współbieżnością z użyciem wielu wersji).

Jeśli baza musi zapewniać tylko odczyt zatwierdzonych danych zamiast izolacji snapshotów, wystarczy przechowywać dwie wersje obiektu: zatwierdzoną i zmodyfikowaną, ale jeszcze niezatwierdzoną. Jednak systemy składowania danych obsługujące izolację snapshotów zwykle posługują się techniką MVCC także do odczytu zatwierdzonych danych. Typowe rozwiązanie polega na tym, że na potrzeby odczytu zatwierdzonych danych dla każdego zapytania używany jest odrębny snapshot, natomiast w izolacji snapshotów w całej transakcji stosowany jest ten sam snapshot.

Na rysunku 7.7 pokazano, jak izolacja snapshotów oparta na technice MVCC jest zaimplementowana w bazie PostgreSQL [31] (inne implementacje wyglądają podobnie). W momencie rozpoczynania transakcji przypisywany jest jej unikatowy, zawsze rosnący<sup>7</sup> identyfikator transakcji (txid). Gdy transakcja zapisuje jakieś dane w bazie, są one opatrywane identyfikatorem transakcji jednostki zapisującej.



Rysunek 7.7. Implementowanie izolacji snapshotów za pomocą wielu wersji obiektów

Każdy wiersz w tabeli obejmuje pole `created_by` zawierające identyfikator transakcji, która wstawiła ten wiersz do tabeli. Ponadto każdy wiersz zawiera początkowo puste pole `deleted_by`. Jeśli transakcja kasuje wiersz, nie jest on usuwany z bazy, ale oznaczony jako przeznaczony do usunięcia. W tym celu do pola `deleted_by` należy przypisać identyfikator transakcji, która zażądała usunięcia wiersza. Jakiś czas później, gdy z pewnością wiadomo, że żadna transakcja nie może uzyskać dostępu do usuniętych danych, proces odzyskiwania pamięci w bazie kasuje wszystkie wiersze przeznaczone do usunięcia i zwalnia zajmowane przez nie miejsce.

<sup>7</sup> Precyzyjnie rzecz biorąc, identyfikatory transakcji to 32-bitowe liczby całkowite. Dlatego po ok. 4 mld transakcji ich numerowanie rozpoczyna się od nowa. Operacja `VACUUM` w bazie PostgreSQL wykonuje operacje porządkujące, co gwarantuje, że rozpoczęcie numerowania od nowa nie wpłynie na dane.

Aktualizacja jest wewnętrznie przekształcana na operacje usuwania i tworzenia. Na przykład na rysunku 7.7 transakcja 13 odejmuje 100 zł z konta 2, zmieniając stan z 500 zł na 400 zł. Tabela accounts zawiera wtedy dwa wiersze dotyczące konta 2: wiersz ze stanem 500 zł oznaczony jako usunięty przez transakcję 13 i wiersz z saldem 400 zł utworzony przez transakcję 13.

### Reguły widoczności pozwalające zobaczyć spójne snapshoty

Gdy transakcja wczytuje dane z bazy, identyfikatory transakcji są używane do ustalenia, które obiekty może ona zobaczyć, a które są dla niej niewidoczne. Dzięki starannemu zdefiniowaniu reguł widoczności baza może udostępnić aplikacji spójny snapshot bazy. Ten mechanizm działa w następujący sposób:

1. Na początku każdej transakcji baza tworzy listę wszystkich innych transakcji wykonywanych w tym czasie (niezatwierdzonych i nieanulowanych). Wszystkie zapisy wprowadzone przez transakcje są ignorowane, nawet jeśli te transakcje zostały później zatwierdzone.
2. Wszystkie zapisy dokonane przez anulowane transakcje są ignorowane.
3. Wszystkie zapisy wprowadzone przez transakcje z późniejszym identyfikatorem (czyli te, które rozpoczęły się po uruchomieniu danej transakcji) są ignorowane niezależnie od tego, czy zostały zatwierdzone.
4. Wszystkie pozostałe zapisy są widoczne w zapytaniach aplikacji.

Te reguły dotyczą zarówno tworzenia obiektów, jak i ich usuwania. Na rysunku 7.7 gdy transakcja 12 wczytuje stan konta 2, widzi saldo 500 zł, ponieważ ten stan został usunięty przez transakcję 13 (a zgodnie z regułą 3. transakcja 12 nie może widzieć skutków usuwania danych przez transakcję 13), a efekty utworzenia salda 400 zł nie są jeszcze widoczne (wynika to z tej samej reguły).

Można to ująć inaczej — obiekt jest widoczny, jeśli spełnione są oba podane niżej warunki:

- W momencie rozpoczęcia transakcji przez jednostkę odczytującą transakcja, która utworzyła dany obiekt, została już zatwierdzona.
- Obiekt nie jest przeznaczony do usunięcia, a jeśli jest, transakcja żądająca skasowania go nie została jeszcze zatwierdzona w momencie uruchomienia transakcji odczytującej.

Długie transakcje mogą korzystać ze snapshota przez długi czas i wciąż wczytywać wartości, które (z perspektywy innych transakcji) już dawno zostały zastąpione lub usunięte. Dzięki temu, że baza nigdy nie aktualizuje wartości w miejscu, a zamiast tego tworzy nową wersję po każdej modyfikacji danych, może udostępniać spójny snapshot niewielkim kosztem.

### Indeksy i izolacja snapshotów

Jak działają indeksy w bazie z wieloma wersjami danych? Jedna możliwość jest taka, by indeks prowadził do wszystkich wersji obiektu, co wymaga zapytania odfiltrowującego wszystkie wersje niewidoczne dla bieżącej transakcji. Gdy mechanizm odzyskiwania pamięci usuwa starsze wersje obiektu, które nie są już widoczne dla żadnej transakcji, powiązane z nimi wpisy w indeksie też można skasować.

W praktyce na wydajność technik MVCC wpływa wiele szczegółów implementacji. Na przykład optymalizacje w bazie PostgreSQL pozwalają uniknąć aktualizowania indeksów, jeśli różne wersje tego samego obiektu można zmieścić na tej samej stronie [31].

Inne podejście jest stosowane w bazach CouchDB, Datomic i LMDB. Choć także w nich używane są b-drzewa (zob. punkt „B-drzewa”), wykorzystuje się tam mechanizm „tylko dołączanie i kopiowanie przy zapisie”. Polega to na tym, że w momencie aktualizacji strony drzewa nie są nadpisywane. Zamiast tego tworzona jest nowa kopia każdej zmodyfikowanej strony. Strony nadrzędne (aż do korzenia drzewa) są kopiowane i aktualizowane, tak by prowadziły do nowych wersji stron podrzędnych. Strony, na które nie wpływa zapis, nie wymagają kopiowania i pozostają bez zmian [33, 34, 35].

W b-drzewach z samym dołączaniem danych każda transakcja zapisu (lub porcja transakcji) tworzy nowy korzeń b-drzewa. Dany korzeń jest spójnym snapshotem bazy z momentu utworzenia go. Nie trzeba wtedy filtrować obiektów na podstawie identyfikatorów transakcji, ponieważ późniejsze zapisy nie mogą zmodyfikować istniejącego b-drzewa — mogą tylko tworzyć nowe korzenie. To podejście wymaga jednak działającego w tle procesu kompresji i odzyskiwania pamięci.

### **Powtarzalny odczyt i problem z terminologią**

Izolacja snapshotów to przydatny poziom izolacji (zwłaszcza w transakcjach tylko wczytujących dane). Jednak w wielu stosujących go bazach nosi on inne nazwy. W Oracle używane jest określenie *sekwencyjność* (ang. *serializable*), a w bazach PostgreSQL i MySQL *powtarzalny odczyt* (ang. *repeatable read*) [23].

Przyczyną tych niejasności w terminologii jest to, że w standardzie SQL-a izolacja snapshotów nie występuje. W tym standardzie wykorzystywana jest definicja poziomów izolacji z bazy System R z 1975 r. [2], a izolacja snapshotów nie została wtedy jeszcze wymyślona. Zamiast tego zdefiniowany jest powtarzalny odczyt, który pozornie wygląda bardzo podobnie do izolacji snapshotów. W bazach PostgreSQL i MySQL izolacja snapshotów jest nazywana powtarzalnym odczytem, ponieważ spełnia wymagania standardu. Dzięki temu twórcy tych baz mogą twierdzić, że ich produkty są zgodne ze standardem.

Niestety, definicja poziomów izolacji w standardzie SQL-a jest błędna. Jest wieloznaczna, nieprecyzyjna i za bardzo zależna od implementacji jak na standard [28]. Choć powtarzalny odczyt jest zaimplementowany w różnych bazach, między zapewnianymi przez nie gwarancjami występują znaczne różnice (mimo że rzekomo są one zgodne ze standardem) [23]. W tekstach naukowych pojawiła się formalna definicja powtarzalnego odczytu [29, 30], jednak większość implementacji nie jest zgodna z tą definicją. A na domiar złego w bazie DB2 firmy IBM „powtarzalny odczyt” oznacza sekwencyjność [8].

W efekcie nikt tak naprawdę nie wie, co oznacza powtarzalny odczyt.

### **Zapobieganie utracie aktualizacji**

Omawiane wcześniej poziomy odczytu zatwierdzonych danych i izolacji snapshotów dotyczą głównie gwarancji tego, co transakcje tylko wczytujące dane zobaczą, gdy jednocześnie dochodzi do zapisu. Nie uwzględniano tam problemu jednoczesnego zapisu danych przez dwie transakcje. Opisano

tylko brudny zapis (zob. punkt „Brak brudnych zapisów”) — jeden konkretny rodzaj konfliktu typu zapis-zapis.

Miedzy transakcjami jednocześnie zapisującymi dane mogą wystąpić także inne ciekawe rodzaje konfliktów. Najbardziej znany z nich to problem *utruty aktualizacji*. Jest on przedstawiony na rysunku 7.1 na przykładzie dwóch jednoczesnych inkrementacji licznika.

Problem utraty aktualizacji może nastąpić, jeśli aplikacja wczytuje jakąś wartość z bazy, modyfikuje ją, a następnie zapisuje zmodyfikowane dane (jest to *cykl wczytaj – zmodyfikuj – zapisz*). Jeśli dwie transakcje robią to jednocześnie, jedna z modyfikacji może zostać utracona, ponieważ drugi zapis nie uwzględnia pierwszej modyfikacji. Można powiedzieć, że wcześniejszy zapis jest *niszczony* przez późniejszy. Ten wzorec występuje w różnych scenariuszach:

- Inkrementacji licznika lub aktualizacji stanu konta (co wymaga wczytania aktualnej wartości, obliczenia nowej i zapisania zaktualizowanych danych).
- Wprowadzania lokalnych zmian w złożonej wartości, np. dodania elementu do listy w dokumencie w formacie JSON (co wymaga parsowania dokumentu, wprowadzenia zmiany i zapisu zmodyfikowanego dokumentu).
- Jednoczesnej edycji strony wiki przez dwóch użytkowników, kiedy to każdy użytkownik zapisuje swoje zmiany przez przesłanie zawartości całej strony na serwer, co powoduje nadpisanie danych obecnie znajdujących się w bazie.

Ponieważ ten problem występuje tak często, opracowano różne rozwiązania.

### Atomowe operacje zapisu

Wiele baz udostępnia atomowe operacje aktualizacji, dzięki czemu nie trzeba implementować cyklu wczytaj – zmodyfikuj – zapisz w kodzie aplikacji. Takie operacje są zwykle najlepszym rozwiązaniem, jeśli można napisać kod przy ich użyciu. Na przykład ta instrukcja jest w większości baz relacyjnych bezpieczna ze względu na współbieżność:

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

Podobnie bazy oparte na dokumentach, np. MongoDB, udostępniają operacje atomowe do wprowadzania lokalnych modyfikacji w części dokumentu w formacie JSON. Baza Redis udostępnia atomowe operacje do modyfikowania struktur danych takich jak kolejki priorytetowe. Nie wszystkie zapisy można łatwo przedstawić w kategoriach operacji atomowych (np. aktualizacje strony wiki obejmują arbitralne zmiany w tekście<sup>8</sup>), jednak w sytuacji, gdy takie operacje można zastosować, są one zwykle najlepszym wyborem.

Operacje atomowe są zazwyczaj implementowane za pomocą blokady wyłącznej obiektu zakładanej na czas odczytu. Wtedy żadna inna transakcja nie może wczytać obiektu do czasu wprowadzenia aktualizacji. To podejście jest czasem nazywane zapewnianiem *stabilności kursora* [36, 37]. Inna możliwość to wymuszenie wykonywania wszystkich operacji atomowych w jednym wątku.

---

<sup>8</sup> Możliwe, choć dość skomplikowane jest przedstawienie edycji dokumentu tekstowego jako strumienia atomowych modyfikacji. Wskazówki znajdziesz w ramce „Automatyczne rozwiązywanie konfliktów”.

Niestety, platformy ORM sprawiają, że można łatwo przypadkowo napisać kod, który wykonuje niebezpieczne cykle wczytaj – zmodyfikuj – zapisz, zamiast stosować udostępniane przez bazę operacje atomowe [38]. Nie stanowi to problemu, jeśli wiesz, co robisz. Jednak potencjalnie jest to źródło subtelnych błędów trudnych do wykrycia w trakcie testów.

## Bezpośrednie używanie blokad

Innym sposobem zapobiegania utracie aktualizacji (jeśli wbudowane operacje atomowe z bazy nie zapewniają potrzebnych funkcji) jest bezpośrednie używanie w aplikacji blokad obiektów, które mają zostać zaktualizowane. Następnie aplikacja może wykonać cykl wczytaj – zmodyfikuj – zapisz, a gdy inna transakcja spróbuje jednocześnie wczytać ten sam obiekt, będzie musiała odczekać do momentu zakończenia wcześniejszego cyklu.

Pomyśl np. o wieloosobowej grze, w której kilku graczy może jednocześnie przesunąć tę samą figurę. W takiej sytuacji operacja atomowa może nie wystarczyć, ponieważ aplikacja musi się też upewnić, że ruch gracza jest zgodny z regułami gry. Wymaga to logiki, której nie da się w sensowny sposób umieścić w zapytaniu kierowanym do bazy. Zamiast tego można posłużyć się blokadą, aby zapobiec jednoczesnemu przenoszeniu tej samej figury przez dwóch graczy. Ilustruje to listing 7.1.

*Listing 7.1. Bezpośrednie blokowanie wierszy, aby zapobiec utracie aktualizacji*

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; ❶

-- Sprawdzenie, czy ruch jest prawidłowy, a następnie aktualizowanie pozycji
-- figury zwróconej przez wcześniejsze polecenie SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

❶ Klauzula `FOR UPDATE` oznacza, że baza powinna objąć blokadą wszystkie wiersze zwrócone przez dane zapytanie.

To rozwiązanie działa, ale aby było poprawne, trzeba starannie przemyśleć logikę aplikacji. Łatwo jest zapomnieć dodać gdzieś w kodzie niezbędną blokadę i spowodować w ten sposób sytuację wyścigu.

## Automatyczne wykrywanie utraconych aktualizacji

Operacje atomowe i blokady pozwalają zapobiec utraconym aktualizacjom dzięki wymuszaniu sekwencyjnego wykonywania cykli wczytaj – zmodyfikuj – zapisz. Inną możliwością to umożliwienie równoległego wykonywania takich cykli i, po wykryciu przez menedżera transakcji utraty aktualizacji, anulowanie transakcji oraz wymuszanie ponowienia przez nią cyklu wczytaj – zmodyfikuj – zapisz.

Zaletą tego podejścia jest to, że bazy mogą wydajnie sprawdzać dane, jeśli stosują jednocześnie izolację snapshotów. Na poziomach powtarzalnych odczytów w bazie PostgreSQL, sekwencyjności w Oracle i izolacji snapshotów w bazie SQL Server następuje automatyczne wykrycie utraty aktu-

alizacji i anulowanie sprawiającej problemy transakcji. Jednak powtarzalny odczyt w bazie MySQL z systemem InnoDB nie zapewnia automatycznego wykrywania utraty aktualizacji [23]. Niektórzy autorzy [28, 30] twierdzą, że baza musi zapobiegać utracie aktualizacji, aby można uznać, że zapewnia izolację snapshotów. Zgodnie z tą definicją baza MySQL nie gwarantuje izolacji snapshotów.

Wykrywanie utraty aktualizacji to bardzo przydatny mechanizm, ponieważ nie wymaga wykorzystywania żadnych specjalnych funkcji bazy w kodzie aplikacji. Możesz zapomnieć o zastosowaniu blokady lub operacji atomowej i spowodować przez to błąd, jednak wykrywanie utraty aktualizacji odbywa się automatycznie, dlatego jest mniej narażone na problemy.

## Porównaj i ustaw

W bazach, które nie zapewniają transakcji, czasem dostępna jest atomowa operacja porównaj i ustaw (wspomniana wcześniej w punkcie „Zapisy pojedynczego obiektu”). Ta operacja ma chronić przed utratą aktualizacji w ten sposób, że aktualizację można wprowadzić tylko, jeśli wartość nie zmieniła się od czasu jej ostatniego odczytu. Jeżeli aktualna wartość nie pasuje do tej wczytanej wcześniej, aktualizacja nie jest wprowadzana i cykl wczytaj – zmodyfikuj – zapisz trzeba ponowić.

Na przykład aby zapobiec jednoczesnej aktualizacji tej samej strony wiki przez dwóch użytkowników, można spróbować następującego rozwiązania (oczekiwane jest wtedy, że aktualizacja nastąpi tylko w sytuacji, gdy zawartość strony nie zmieniła się od czasu rozpoczęcia jej edycji przez użytkownika):

```
-- Ta operacja może być bezpieczna, ale nie musi. Zależy to od implementacji bazy danych
UPDATE wiki_pages SET content = 'nowa zawartość'
WHERE id = 1234 AND content = 'dawna zawartość';
```

Jeśli zawartość strony się zmieniła i nie jest nią już 'dawna zawartość', aktualizacja nie zostanie wprowadzona. Dlatego trzeba sprawdzać, czy aktualizacja się powiodła, i w razie potrzeby ponawiać próby. Jeżeli jednak baza umożliwia klauzuli WHERE odczyt danych z dawnego snapshota, ta instrukcja może nie zapobiegać utracie aktualizacji, ponieważ warunek jest spełniony mimo tego, że nastąpił inny jednoczesny zapis. Zanim zaczniesz polegać na operacji porównaj i ustaw, sprawdź, czy jest ona bezpieczna w bazie, z której korzystasz.

## Rozwiązywanie konfliktów i replikacja

W bazach z replikacją (zob. rozdział 5.) zapobieganie utracie aktualizacji wkracza w inny wymiar. Ponieważ kopie danych znajdują się w wielu węzłach, a dane mogą być w nich jednocześnie modyfikowane, to aby zapobiec utracie aktualizacji, trzeba podjąć dodatkowe kroki.

W blokadach oraz operacjach porównaj i ustaw przyjmuje się, że istnieje jedna aktualna kopia danych. Jednak bazy z replikacją z wieloma liderami lub bez lidera zwykle umożliwiają jednoczesne wprowadzanie wielu zapisów i ich asynchroniczną replikację. Nie mogą więc zagwarantować istnienia jednej aktualnej kopii danych. Dlatego w tym kontekście techniki oparte na blokadach oraz operacji porównaj i ustaw nie mają zastosowania. Szczegółowe omówienie tej kwestii znajdziesz w punkcie „Linijowość”.



Zamiast tego, jak opisano w punkcie „Wykrywanie jednoczesnych zapisów”, typowe podejście w bazach z replikacją polega na umożliwieniu jednoczesnym zapisom utworzenia kilku niezgodnych wersji wartości (*wartości siostrzanych*) i wykorzystaniu kodu aplikacji lub specjalnych struktur danych do późniejszego uzgadniania i scalania tych wersji.

Operacje atomowe mogą działać poprawnie w kontekście z replikacją, zwłaszcza jeśli są przemienne (tzn. że można je stosować w innej kolejności w różnych replikach i uzyskać ten sam wynik). Na przykład inkrementacja licznika lub dodanie elementu do zbioru to operacje przemienne. Zgodnie z tym podejściem działają typy danych z systemu Riak 2.0, które zapobiegają utracie aktualizacji na poziomie wielu replik. Gdy wartość jest jednocześnie aktualizowana przez różne klienty, Riak automatycznie scala aktualizacje w taki sposób, że żadna z nich nie zostaje utracona [39].

Z kolei metoda eliminowania konfliktów *ostatni zapis wygrywa* jest narażona na utratę aktualizacji, co opisano w punkcie „Ostatni zapis wygrywa (odrzuć jednoczesnych zapisów)”. Niestety, w wielu bazach z replikacją domyślnie stosowana jest właśnie metoda ostatni zapis wygrywa.

## Zapis zniekształcający i fantomy

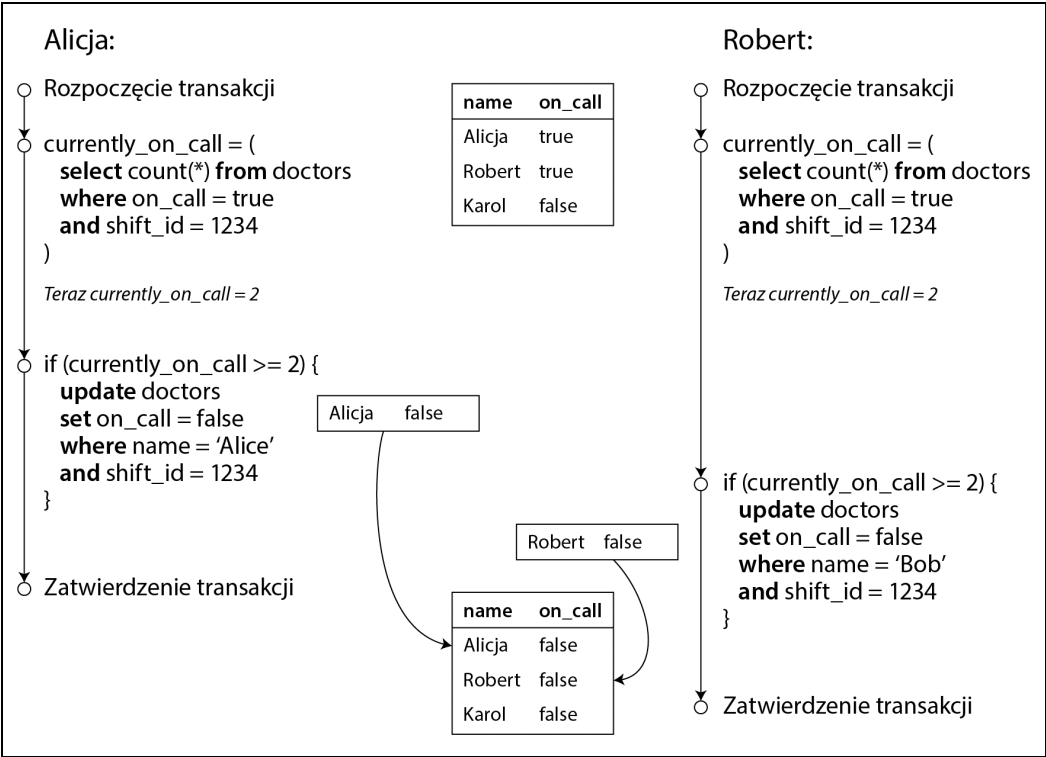
We wcześniejszych punktach opisano *brudne zapisy* i *utratę aktualizacji* — dwa rodzaje sytuacji wyścigu, które mogą nastąpić, gdy różne transakcje jednocześnie próbują zapisywać dane w tych samych obiektach. Aby uniknąć uszkodzenia danych, trzeba zapobiegać sytuacji wyścigu — albo automatycznie w bazie, albo za pomocą ręcznych zabezpieczeń, np. stosując blokady lub atomowe operacje zapisu.

To jednak nie koniec listy potencjalnych sytuacji wyścigu związanych z jednoczesnymi zapisami. W tym punkcie opisane są bardziej subtelne rodzaje konfliktów.

Na początku wyobraź sobie następujący przykład: piszesz przeznaczoną dla lekarzy aplikację do zarządzania dyżurami w szpitalu. Zarząd szpitala zwykle stara się, aby w każdej chwili dostępnych na wezwanie było kilku doktorów. Jednak zawsze dyżuować musi przynajmniej jeden lekarz. Doktorzy mogą zrezygnować ze swoich zmian (np. jeśli się rozchorują), ale pod warunkiem że na danym dyżurze pozostanie przynajmniej jeden z nich [40, 41].

Teraz wyobraź sobie, że Alicja i Robert to dwójka dyżurnych lekarzy na danej zmianie. Oboje czują się źle, dlatego decydują się zażądać wolnego. Niestety, oboje klikają przycisk rezygnacji z dyżuru mniej więcej w tym samym czasie. Dalszy rozwój wypadków jest przedstawiony na rysunku 7.8.

W każdej transakcji aplikacja najpierw sprawdza, czy do dyżuru przypisanych jest obecnie przynajmniej dwóch lekarzy. Jeśli tak, aplikacja przyjmuje, że można bezpiecznie pozwolić jednemu doktorowi na rezygnację z dyżuru. Ponieważ w bazie używa się izolacji snapshotów, w obu testach zwracana wartość to 2. Dlatego obie transakcje przechodzą do następnego etapu. Alicja aktualizuje własny rekord, rezygnując z dyżuru. Robert wykonuje taką samą operację. Obie transakcje są zatwierdzane, przez co na zmianie nie będzie żadnego lekarza. Wymóg przynajmniej jednego doktora na dyżurze został więc złamany.



Rysunek 7.8. Przykład zapisu zniekształcającego powodującego błąd w aplikacji

### Cechy zapisu zniekształcającego

Ta anomalia to *zapis zniekształcający* (ang. *write skew*) [28]. To ani brudny zapis, ani utrata aktualizacji, ponieważ obie transakcje aktualizują dwa różne obiekty (rekordy dyżurów Alicji i Roberta). Konflikt jest tu mniej oczywisty, jednak bez wątpienia występuje sytuacja wyścigu. Gdyby obie transakcje działały jedna po drugiej, rezygnacja drugiego lekarza z dyżuru byłaby niemożliwa. Ta anomalia mogła nastąpić tylko z powodu jednoczesnego wykonywania transakcji.

Możesz potraktować zapis zniekształcający jako uogólnienie problemu utraty aktualizacji. Zapis zniekształcający może wystąpić, gdy dwie transakcje wczytują te same obiekty, a następnie aktualizują niektóre z nich (różne transakcje mogą tu aktualizować odmienne obiekty). W specjalnym przypadku, gdy różne transakcje aktualizują ten sam obiekt, dochodzi do brudnego zapisu lub utraty aktualizacji (zależy to od rozkładu zdarzeń w czasie).

Zobaczyłeś, że istnieje wiele różnych sposobów zapobiegania utracie aktualizacji. Przy zapisie zniekształcającym tych możliwości jest mniej:

- Atomowe operacje na jednym obiekcie nie są pomocne, ponieważ używanych jest wiele obiektów.
- Automatyczne wykrywanie utraty aktualizacji dostępne w niektórych implementacjach izolacji snapshotów niestety też nie pomaga. Zapis zniekształcający nie jest automatycznie wykrywany w powtarzalnym odczycie w bazie PostgreSQL, w powtarzalnym odczycie w bazie MySQL

z systemem InnoDB, w podejściu sekwencyjnym w bazie Oracle czy w izolacji snapshotów w bazie SQL Server [23]. Automatyczne zapobieganie zapisowi zniekształcającemu wymaga prawdziwej izolacji z użyciem sekwencyjności (zob. punkt „Sekwencyjność”).

- Niektóre bazy umożliwiają skonfigurowanie więzów, które są później wymuszane przez bazę (np. więzów unikatowości lub klucza obcego albo ograniczeń konkretnej wartości). Jednak aby móc określić, że przynajmniej jeden lekarz musi pozostać na dyżurze, potrzebne byłyby więzy dotyczące wielu obiektów. Większość baz nie zapewnia wbudowanej obsługi takich więzów, choć w niektórych bazach możesz je zaimplementować za pomocą wyzwalaczy lub widoków zmaterializowanych [42].
- Jeśli nie możesz zastosować sekwencyjności, drugą najlepszą możliwością są prawdopodobnie bezpośrednie blokady wierszy potrzebnych w transakcji. W przykładzie z lekarzami możesz napisać kod podobny do poniższego:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; ❶

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alicja'
  AND shift_id = 1234;

COMMIT;
```

❶ Tak jak wcześniej instrukcja `FOR UPDATE` informuje bazę, by zablokowała wszystkie wiersze zwrócone przez zapytanie.

## Więcej przykładów zapisu zniekształcającego

Zapis zniekształcający początkowo może się wydawać rzadkim problemem, jednak gdy o nim pomyślisz, możliwe, że zauważysz więcej sytuacji, w jakich może występować. Oto kilka innych przykładów:

### System rezerwacji sali konferencyjnej

Założmy, że chcesz zapewnić, by nie można było dodać dwóch rezerwacji tej samej sali konferencyjnej na tę samą godzinę [43]. Gdy ktoś chce zarezerwować salę, najpierw należy poszukać niezgodnych rezerwacji (dotyczących tej samej sali w pokrywającym się czasie). Jeśli niezgodne rezerwacje nie istnieją, można dodać spotkanie (zob. listing 7.2)<sup>9</sup>.

*Listing 7.2. System rezerwacji sali spotkań próbuje uniknąć podwójnych rezerwacji (na poziomie izolacji snapshotów nie jest to bezpieczne rozwiązanie)*

```
BEGIN TRANSACTION;

-- Wyszukiwanie istniejących rezerwacji nachodzących na czas od 12:00 do 13:00
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
  end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';
```

---

<sup>9</sup> W bazie PostgreSQL można to zrobić w bardziej elegancki sposób, wykorzystując typy zakresowe, które jednak w innych bazach nie są powszechnie dostępne.

```
-- Jeśli poprzednie zapytanie zwróciło zero:
INSERT INTO bookings
(room_id, start_time, end_time, user_id)
VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

Niestety, izolacja snapshotów nie zabezpiecza przed jednoczesnym dodaniem powodującego konflikt spotkania przez innego użytkownika. Aby zagwarantować brak konfliktów, także tu trzeba zapewnić sekwencyjność.

### *Gra wieloosobowa*

Na listingu 7.1 do zapobiegania utracie aktualizacji zastosowano blokadę (co gwarantuje, że dwóch graczy nie może jednocześnie przesunąć tej samej figury). Jednak blokada nie zapobiega przesunięciu przez różnych graczy dwóch innych figur w to samo miejsce planszy lub wykonaniu innych ruchów naruszających zasady gry. W zależności od rodzaju wymuszanej reguły czasem można zastosować więzy unikatowości, jednak w innych sytuacjach będziesz narażony na zapis zniekształcający.

### *Zajmowanie nazwy użytkownika*

W witrynie, w której każdy użytkownik ma unikatową nazwę, dwie osoby mogą próbować jednocześnie utworzyć konto z tą samą nazwą użytkownika. Możesz posłużyć się transakcją do sprawdzania, czy dana nazwa jest już zajęta, i tworzenia konta o tej nazwie, jeśli jest ona wolna. Jednak tak jak w poprzednich przykładach, nie jest to bezpieczne, gdy używa się izolacji snapshotów. Na szczęście prostym rozwiązaniem są tu więzy unikatowości (druga transakcja próbująca zarejestrować daną nazwę zostanie anulowana z powodu naruszenia więzów).

### *Zapobieganie dwukrotnemu wydatkowaniu*

Usługa umożliwiająca użytkownikom wydawanie pieniędzy lub punktów musi sprawdzać, czy użytkownik nie wyda więcej, niż posiada. Możesz zaimplementować to rozwiązanie, dodając próbnie do konta użytkownika operację wydającą środki, tworząc listę wszystkich operacji konta i sprawdzając, czy suma jest dodatnia [44]. Zapis zniekształcający może spowodować, że dwie operacje wydające środki zostaną dodane jednocześnie, co doprowadzi do ujemnego stanu konta, ale żadna z tych transakcji nie wykryje drugiej.

### **Zapis zniekształcający z powodu wartości fantomowych**

We wszystkich opisanych przykładach wzorec jest podobny:

1. Zapytanie SELECT sprawdza, czy dane wymaganie jest spełnione. W tym celu szuka wierszy spełniających określony warunek wyszukiwania (na dyżurze jest przynajmniej dwóch lekarzy, nie istnieją rezerwacje danej sali w wyznaczonym czasie, na polu planszy nie znajduje się inna figura, nazwa użytkownika nie jest jeszcze zajęta, na koncie znajdują się środki).
2. W zależności od wyniku pierwszego zapytania kod aplikacji decyduje, co robić dalej (albo wykonuje działania, albo zgłasza błąd użytkownikowi i anuluje operację).
3. Jeśli aplikacja uznaje, że należy kontynuować pracę, wykonuje zapis w bazie (operacja INSERT, UPDATE lub DELETE) i zatwierdza transakcję.

Efekt zapisu zmienia warunek wstępny związany z decyzją z kroku 2. Oznacza to, że jeśli po zatwierdzeniu zapisu ponownie wykonasz operację SELECT z kroku 1., otrzymasz inny wynik, ponieważ zapis zmienił zestaw wierszy pasujących do warunku wyszukiwania (teraz na dyżurze jest o jednego lekarza mniej, sala spotkań jest zarezerwowana na daną godzinę, na polu planszy znajduje się przesunięta tam figura, nazwa użytkownika jest już zajęta, na koncie dostępnych jest mniej pieniędzy).

Te kroki mogą być wykonywane w różnej kolejności. Najpierw możesz przeprowadzić zapis, potem wykonać zapytania SELECT, a w ostatnim kroku na podstawie wyniku zapytania zdecydować, czy anulować, czy zatwierdzić transakcję.

W scenariuszu dotyczącym lekarzy na dyżurze wiersz modyfikowany w kroku 3. jest jednym z wierszy zwracanych w kroku 1. Dlatego można zapewnić bezpieczeństwo transakcji i uniknąć zapisu zniekształcającego, blokując wiersze w kroku 1. (SELECT FOR UPDATE). Jednak cztery pozostałe sytuacje są inne. Sprawdzają *brak* wierszy pasujących do warunku wyszukiwania, a zapis *dodaje* wiersz pasujący do tego samego warunku. Jeśli zapytanie w kroku 1. nie zwraca żadnych wierszy, instrukcja SELECT FOR UPDATE nie może zastosować blokady do żadnych danych.

Ten efekt polegający na tym, że zapis w jednej transakcji zmienia wynik wyszukiwania z innej transakcji, jest nazywany *fantomem* [3]. Izolacja snapshotów pozwala uniknąć fantomów w zapytaniach tylko wczytujących dane, jednak w transakcjach wykonujących odczyt i zapis (takich jak w opisanych scenariuszach) fantomy mogą prowadzić do skomplikowanych przypadków zapisu zniekształcającego.

## Materializowanie konfliktów

Skoro problem z fantomami polega na tym, że nie istnieje obiekt, do którego można zastosować blokadę, może warto sztucznie dodać blokowany obiekt do bazy?

Na przykład w scenariuszu z rezerwacją sali konferencyjnej można sobie wyobrazić tabelę z pomieszczeniami i oknami czasowymi. Każdy wiersz tej tabeli odpowiada konkretnej sali w określonym czasie (np. dla 15 minut). Można z góry utworzyć wiersze dla wszystkich możliwych kombinacji sal i okien czasowych np. na sześć miesięcy wprzód.

Teraz transakcja, która chce dodać rezerwację, może zablokować (za pomocą instrukcji SELECT FOR UPDATE) wiersze tabeli odpowiadające potrzebnej kombinacji sali i okien czasowych. Po zajęciu blokady można sprawdzić, czy nie występują sprzeczne rezerwacje, i dodać nową rezerwację w opisany wcześniej sposób. Zauważ, że dodatkowa tabela nie służy do przechowywania informacji o rezerwacjach. Stanowi ona wyłącznie kolekcję blokad używanych do zapobiegania jednoczesnym modyfikacjom rezerwacji tej samej sali w tym samym czasie.

To podejście bywa nazywane *materializowaniem konfliktów*, ponieważ przekształca fantomy w konflikt związany z blokadą konkretnego zestawu wierszy z bazy [11]. Niestety, ustalanie tego, jak materializować konflikty, może być trudne i narażone na błędy. Ponadto wprowadzanie mechanizmu zarządzania współbieżnością do modelu danych aplikacji jest bardzo nieeleganckie. Z tych powodów materializowanie konfliktów należy traktować jak ostatnią deskę ratunku, jeśli żadne inne rozwiązanie nie jest możliwe. W większości sytuacji dużo bardziej wskazane jest zapewnienie sekwencyjności.

# Sekwencyjność

W tym rozdziale przyjrzałeś się kilku przykładowym transakcjom narażonym na sytuację wyścigu. Niektórym sytuacjom wyścigu można zapobiec dzięki dwóm poziomom izolacji — odczytowi zatwierdzonych danych i izolacji snapshotów. Jednak innych problemów te poziomy nie rozwiązują. Zobaczyłeś tu wyjątkowo skomplikowane scenariusze z zapisem zniekształcającym i fantomami. Jest to niekorzystna sytuacja:

- Poziomy izolacji są trudne do zrozumienia i niespójnie implementowane w różnych bazach (np. „powtarzalny odczyt” może oznaczać bardzo różne rzeczy).
- Na podstawie kodu aplikacji trudno jest stwierdzić, czy można bezpiecznie używać danego poziomu izolacji. Dotyczy to zwłaszcza rozbudowanych aplikacji, w których możesz nie być świadomy wszystkich operacji wykonywanych w tym samym czasie.
- Nie istnieją dobre narzędzia pomagające wykrywać sytuację wyścigu. Teoretycznie pomocne mogą być statyczne analizy [26], jednak techniki opisywane w badaniach nie trafiły jeszcze do praktycznego użytku. Testowanie problemów ze współbieżnością jest trudne, ponieważ są one zwykle niedeterministyczne — trudności pojawiają się tylko w sytuacji nieszczęśliwego zbiegu zdarzeń w czasie.

Nie jest to nowy problem. Dzieje się tak od lat 70., kiedy to po raz pierwszy wprowadzono niskie poziomy izolacji [2]. Odpowiedź badaczy od zawsze brzmi prosto: należy stosować izolację na poziomie *sekwencyjności*!

Sekwencyjność bywa zwykle uznawana za najwyższy poziom izolacji. Gwarantuje, że choć transakcje mogą być wykonywane równolegle, efekt końcowy jest taki sam jak w sytuacji, gdy są uruchamiane jedna po drugiej, *sekwencyjnie*, bez współbieżności. Baza zapewnia w ten sposób, że jeśli transakcje działają prawidłowo, gdy są wykonywane pojedynczo, będą poprawne także, gdy zaczną działać jednocześnie. Oznacza to, że baza zapobiega *wszystkim* możliwym sytuacjom wyścigu.

Skoro jednak sekwencyjność jest korzystniejsza od bałaganu z niskimi poziomami izolacji, dlaczego nie wszyscy z niej korzystają? Aby odpowiedzieć na to pytanie, trzeba się przyjrzeć sposobom implementowania sekwencyjności i działaniu takich rozwiązań. Obecnie większość baz zapewniających sekwencyjność stosuje jedną z trzech technik analizowanych w kolejnych punktach tego rozdziału:

- Bezpośrednie wykonywanie transakcji w porządku sekwencyjnym (zob. punkt „Rzeczywiste sekwencyjne wykonywanie transakcji”).
- Blokady dwuetapowe (zob. punkt „Blokady dwuetapowe”), które przez kilka dziesięcioleci były jednym sensownym rozwiązaniem.
- Optymistyczne techniki zarządzania współbieżnością, np. algorytm SSI (zob. punkt „Algorytm SSI”).

Na razie te techniki zostaną omówione głównie w kontekście baz działających w jednym węźle. W rozdziale 9. pokazano, jak uogólnić te techniki na transakcje dotyczące wielu węzłów w systemie rozproszonym.

## Rzeczywiste sekwencyjne wykonywanie transakcji

Najprostszym sposobem unikania problemów ze współbieżnością jest całkowite jej wyeliminowanie i wykonywanie tylko jednej transakcji naraz, w porządku sekwencyjnym, w jednym wątku. To pozwala całkowicie wyeliminować problem wykrywania konfliktów między transakcjami i zapobiegania im. Uzyskana izolacja jest z definicji sekwencyjna.

Choć to rozwiązanie może się wydawać oczywiste, projektanci baz danych dopiero niedawno (ok. 2007 r.) zdecydowali, że możliwe jest wykorzystanie jednowątkowej pętli do wykonywania transakcji [45]. Skoro współbieżność z użyciem wielu wątków była przez wcześniejsze 30 lat uznawana za niezbędną do uzyskania wysokiej wydajności, co się zmieniło, że wykonywanie transakcji w jednym wątku stało się wykonalne?

Ta zmiana nastawienia wynika z dwóch nowości:

- Pamięć RAM stała się na tyle tania, że w wielu scenariuszach możliwe jest przechowywanie całego aktywnego zbioru danych w pamięci (zob. punkt „Utrzymywanie wszystkiego w pamięci”). Gdy wszystkie dane potrzebne transakcji znajdują się w pamięci, transakcje mogą być wykonywane znacznie szybciej niż wtedy, gdy transakcja musi czekać na wczytanie danych z dysku.
- Projektanci baz danych zauważyli, że transakcje OLTP są zwykle krótkie i wykonują niewielką liczbę odczytów oraz zapisów (zob. punkt „Przetwarzanie transakcji i analityka”). Z kolei długie zapytania analityczne zwykle tylko wczytują dane, dlatego można je uruchamiać z użyciem spójnego snapshota (z izolacją snapshotów) poza pętlą sekwencyjnego wykonywania transakcji.

Sekwencyjne wykonywanie transakcji jest zaimplementowane w bazach VoltDB/H-Store, Redis i Datomic [46, 47, 48]. System zaprojektowany pod kątem wykonywania w jednym wątku czasem działa wydajniej niż system obsługujący współbieżność, ponieważ unika ponoszenia kosztów koordynowania blokad. Jednak jego przepustowość jest ograniczona do poziomu jednego rdzenia procesora. Aby optymalnie wykorzystać jeden wątek, transakcje trzeba ustrukturyzować inaczej niż w tradycyjnym podejściu.

### Umieszczanie transakcji w procedurach składowanych

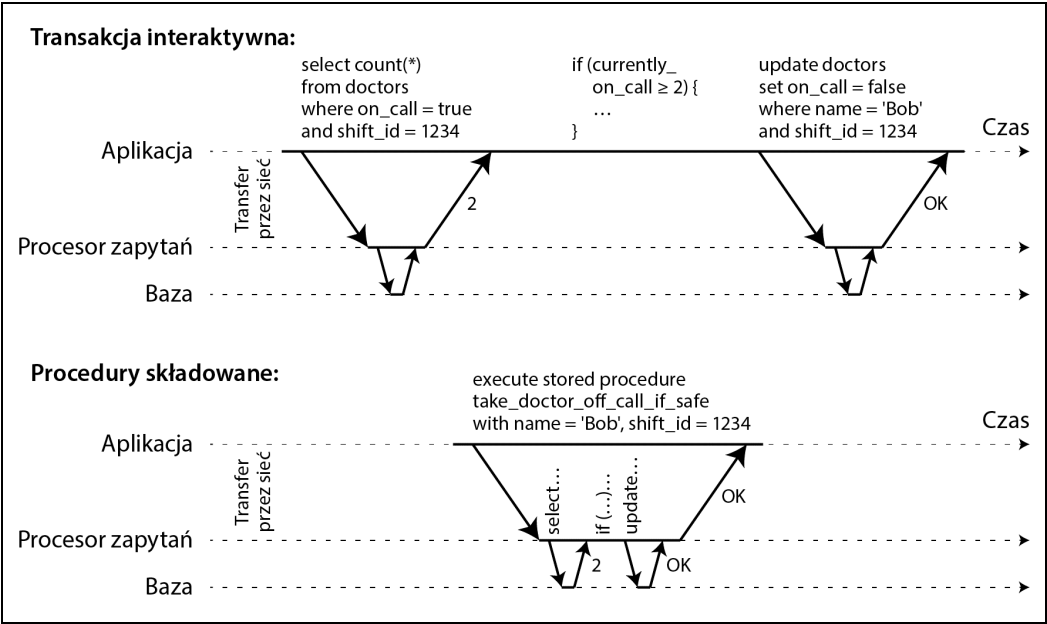
Gdy bazy danych dopiero powstawały, transakcje miały obejmować cały przepływ działań użytkowników. Na przykład rezerwowanie biletu lotniczego to proces wieloetapowy (obejmujący wyszukiwanie tras, opłat i dostępnych miejsc; określanie planu podróży; rezerwowanie miejsc w każdym locie z planu podróży; wprowadzania szczegółowych informacji o pasażerze; dokonywaniu płatności). Projektanci baz danych uważali, że zgrabnym rozwiązaniem byłoby umieszczenie całego procesu w jednej transakcji, aby można go było zatwierdzić atomowo.

Niestety, ludzie podejmują decyzje i reagują bardzo powoli. Jeśli transakcja w bazie ma oczekiwać na dane wejściowe od użytkownika, baza musi potencjalnie obsługiwać dużą liczbę jednoczesnych transakcji, z których większość jest nieaktywna. Większość baz nie potrafi robić tego w wydajny sposób, dlatego prawie wszystkie aplikacje OLTP starają się tworzyć krótkie transakcje, unikając interaktywnego oczekiwania na użytkownika w transakcji. W internecie oznacza to, że transakcja jest zatwierdzana w ramach tego samego żądania HTTP. Transakcja nie rozciąga się więc na wiele żądań. Nowe żądanie HTTP rozpoczyna nową transakcję.

Choć udało się wyeliminować ludzi ze ścieżki krytycznej, transakcje nadal były wykonywane w interaktywnym modelu klient-serwer jedna instrukcja po drugiej. Aplikacja zgłasza zapytanie, wczytuje wynik, możliwe, że uruchamia następne zapytanie na podstawie wyniku pierwszego itd. Zapytania i wyniki są przesyłane tam i z powrotem między kodem aplikacji (działającym na jednej maszynie) a serwerem bazodanowym (z innej maszyny).

W interaktywnych transakcjach dużą część czasu zajmuje komunikacja sieciowa między aplikacją a bazą. Gdyby zrezygnować ze współbieżności w bazie i przetwarzać tylko jedną transakcję po drugiej, przepustowość byłaby bardzo niska, ponieważ baza większość czasu spędzałaby na oczekiwaniu na to, aż aplikacja zgłosi następne zapytanie dotyczące bieżącej transakcji. Aby uzyskać akceptowalną wydajność w bazie tego rodzaju, konieczne jest równoległe przetwarzanie wielu transakcji.

Dlatego systemy z sekwencyjnym przetwarzaniem transakcji w jednym wątku nie pozwalają na wykonywanie interaktywnych transakcji z wieloma instrukcjami. Zamiast tego aplikacja musi z góry przesłać cały kod transakcji do bazy jako *procedurę składowaną*. Różnice między tymi podejściami są przedstawione na rysunku 7.9. Jeśli wszystkie dane potrzebne w transakcji znajdują się w pamięci, procedura składowana może działać bardzo szybko, bez oczekiwania na sieciowe lub dyskowe operacje wejścia-wyjścia.



Rysunek 7.9. Różnica między transakcją interaktywną a procedurą składowaną (na przykładzie transakcji z rysunku 7.8)

### Wady i zalety procedur składowanych

Procedury składowane są dostępne od pewnego czasu w bazach relacyjnych i są częścią standardu SQL-a (SQL/PSM) od 1999 r. Z różnych przyczyn zyskały nie najlepszą reputację:



- Każdy producent baz danych stosuje własny język dla procedur składowanych (w bazach Oracle jest to PL/SQL, w bazach SQL Server jest to T-SQL, w bazach PostgreSQL — PL/pgSQL itd.). Te języki nie nadążały za rozwojem ogólnych języków programowania, dlatego z dzisiejszej perspektywy wyglądają nieelegancko i archaicznie. Nie mają też ekosystemu bibliotek dostępnych w większości języków programowania.
- Kodem działającym w bazie trudno jest zarządzać. W porównaniu z kodem z serwera aplikacji trudniej jest go debugować, obejmować kontrolą wersji, wdrażać, testować i integrować z systemem rejestrowania wskaźników na potrzeby monitorowania.
- W bazie wydajność jest często dużo ważniejsza niż na serwerze aplikacji, ponieważ jedna instancja bazy nieraz jest współużytkowana przez wiele takich serwerów. Źle napisana procedura składowana (zajmująca dużo pamięci lub czasu procesora) może spowodować więcej problemów niż równie źle napisany kod z serwera aplikacji.

Jednak z takimi problemami można sobie poradzić. W nowych implementacjach procedur składowanych zrezygnowano z języka PL/SQL. Zamiast niego używane są istniejące języki do ogólnego użytku. W bazach VoltDB używane są Java lub Groovy, w bazach Datomic Java lub Clojure, a w bazach Redis — Lua.

Dzięki procedurom składowanym i danym przechowywanym w pamięci wykonywanie wszystkich transakcji w jednym wątku staje się wykonalne. Ponieważ nie trzeba wtedy oczekiwać na operacje wejścia-wyjścia i można uniknąć kosztów stosowania innych mechanizmów zarządzania współbieżnością, możliwe jest uzyskanie wysokiej przepustowości w jednym wątku.

W bazach VoltDB procedury składowane stosuje się też na potrzeby replikacji. Zamiast kopiować wykonywane przez transakcję zapisy z jednego wątku do drugiego, w każdej replice wykonuje się tę samą procedurę składowaną. Dlatego w bazach VoltDB niezbędne są *deterministyczne* procedury składowane (które uruchamiane w różnych węzłach, muszą zwracać ten sam wynik). Jeśli transakcja potrzebuje wykorzystać bieżącą datę i aktualny czas, musi to zrobić za pomocą specjalnych deterministycznych interfejsów API.

## Podział na partycje

Dzięki sekwencyjnemu wykonywaniu wszystkich transakcji zarządzanie współbieżnością jest znacznie łatwiejsze. Ogranicza to jednak przepustowość transakcji w bazie do szybkości jednego rdzenia procesora z jednej maszyny. Transakcje tylko wczytujące dane można wykonywać w innych miejscach, używając izolacji snapshotów. Jednak w aplikacjach o dużej liczbie zapisów jednowątkowy procesor transakcji może się stać wąskim gardłem.

Aby możliwe było skalowanie tego rozwiązania z użyciem rdzeni procesora i wielu węzłów, można podzielić dane na partycje (zob. rozdział 6.). Baza VoltDB obsługuje to rozwiązanie. Jeśli znajdziesz sposób podziału zbioru danych w taki sposób, by każda transakcja potrzebowała tylko wczytywać i zapisywać dane z jednej partycji, wtedy każda partycja będzie mogła używać własnego wątku przetwarzania transakcji działającego niezależnie od pozostałych. W takim scenariuszu możesz przypisać każdemu rdzeniowi procesora odrębną partycję. Dzięki temu przepustowość transakcji może rosnąć liniowo względem liczby rdzeni procesora [47].

Jednak w transakcji, która wymaga dostępu do wielu partycji, baza musi koordynować pracę transakcji we wszystkich używanych partycjach. Procedura składowana musi być wykonywana jednocześnie we wszystkich partycjach, aby zagwarantować sekwencyjność w ramach całego systemu.

Ponieważ transakcje obejmujące wiele partycji wymagają dodatkowych kosztów związanych z koordynacją, są znacznie wolniejsze niż transakcje używające jednej partycji. Twórcy bazy VoltDB deklarują przepustowość 1000 zapisów w różnych partycjach na sekundę. Ten wynik jest o kilka rzędów wielkości niższy niż przepustowość jednej partycji i nie da się go zwiększyć przez dodanie kolejnych maszyn [49].

To, czy transakcji może wystarczać jedna partycja, w bardzo dużym stopniu zależy od struktury używanych w aplikacji danych. Proste dane typu klucz-wartość często można bardzo łatwo podzielić. Jednak dane z wieloma indeksami pomocniczymi zwykle wymagają rozbudowanej koordynacji między partycjami (zob. punkt „Podział na partycje i indeksy pomocnicze”).

### Podsumowanie sekwencyjnego wykonywania transakcji

Sekwencyjne wykonywanie transakcji stało się wykonalnym sposobem zapewniania izolacji na poziomie sekwencyjności, choć z pewnymi ograniczeniami:

- Każda transakcja musi być niewielka i szybka, ponieważ wystarczy jedna powolna transakcja, aby zablokować przetwarzanie wszystkich transakcji.
- Technika ta jest ograniczona do scenariuszy, w których aktywny zbiór danych można zmieścić w pamięci. Rzadko używane dane można potencjalnie przenieść na dysk, jeśli jednak potrzebny jest dostęp do nich w transakcji działającej w jednym wątku, system będzie działał bardzo powoli<sup>10</sup>.
- Częstotliwość zapisów musi być na tyle niska, by można je było obsłużyć za pomocą jednego rdzenia procesora. W przeciwnym razie transakcje wymagają podziału w taki sposób, by nie była konieczna koordynacja pracy na poziomie wielu partycji.
- Transakcje obejmujące różne partycje są wykonalne, ale w ograniczonym zakresie.

## Blokady dwuetapowe

Przez ok. 30 lat istniał tylko jeden powszechnie stosowany algorytm zapewniania sekwencyjności w bazach danych: *blokady dwuetapowe* (ang. *two-phase locking* — **2PL**)<sup>11</sup>.



### Blokady dwuetapowe to nie to samo co zatwierdzanie dwuetapowe

Zauważ, że choć *blokady dwuetapowe* mogą się kojarzyć z *zatwierdzaniem dwuetapowym* (ang. *two-phase commit* — **2PC**), są to zupełnie inne rzeczy. Zatwierdzanie dwuetapowe jest opisane w rozdziale 9.

<sup>10</sup> Jeśli transakcja potrzebuje dostępu do danych spoza pamięci, najlepszym rozwiązaniem może być anulowanie transakcji, asynchroniczne pobranie danych do pamięci w trakcie przetwarzania innych transakcji i późniejsze ponowne uruchomienie transakcji po wczytaniu danych. To podejście to *unikanie pamięci podręcznej* i zostało wspomniane w punkcie „Przechowywanie wszystkich danych w pamięci”.

<sup>11</sup> Czasem są one nazywane *ścisłymi blokadami dwuetapowymi* (ang. *strong strict two-phase locking* — **SS2PL**), aby odróżnić je od innych postaci blokad dwuetapowych.

Wcześniej zobaczyłeś, że blokady często są stosowane do zapobiegania brudnym zapisom (zob. punkt „Brak brudnych zapisów”). Jeśli dwie transakcje jednocześnie próbują zapisać dane w tym samym obiekcie, blokada gwarantuje, że druga jednostka zapisująca przed kontynuowaniem działania musi poczekać na zakończenie pracy (anulowanie lub zatwierdzenie operacji) przez pierwszą.

Blokady dwuetapowe działają podobnie, przy czym wymóg związany z blokadą jest znacznie mocniejszy. Kilka transakcji może jednocześnie wczytywać ten sam obiekt, o ile nikt nie zapisuje w nim danych. Jednak gdy jakaś jednostka próbuje zapisywać dany obiekt (modyfikować go lub usuwać), wymagany jest dostęp w trybie na wyłączność:

- Jeśli transakcja A wczytała obiekt, a transakcja B próbuje go zapisać, B musi poczekać na zatwierdzenie lub anulowanie operacji przez transakcję A; dopiero potem B może kontynuować pracę. To gwarantuje, że B nie zmodyfikuje nieoczekiwanie obiektu „za plecami” transakcji A.
- Jeżeli transakcja A zapisała obiekt i transakcja B chce go wczytać, B musi poczekać na zatwierdzenie lub anulowanie operacji przez B; dopiero potem B może kontynuować działanie. Przy blokadach dwuetapowych odczyt starej wersji obiektu, jak pokazano na rysunku 7.1, jest nieakceptowalny.

Przy blokadach dwuetapowych jednostka zapisująca blokuje nie tylko inne takie komponenty, ale też jednostki wczytujące dane (i na odwrót). W izolacji snapshotów obowiązuje podejście *jednostki wczytujące nigdy nie blokują jednostek zapisujących, a jednostki zapisujące nigdy nie blokują jednostek wczytujących* (zob. punkt „Implementowanie izolacji snapshotów”). To podejście ujmuje podstawową różnicę między izolacją snapshotów a blokadami dwuetapowymi. Ponieważ blokady dwuetapowe zapewniają sekwencyjność, chronią przed wszystkimi opisanymi wcześniej sytuacjami wyścigu, w tym przed utratą aktualizacji i zapisem zniekształcającym.

## Implementowanie blokad dwuetapowych

Blokady dwuetapowe są stosowane do izolacji na poziomie sekwencyjności w bazach MySQL (z systemem InnoDB) i SQL Server oraz na poziomie powtarzalnych odczytów w bazach DB2 [23, 36].

Blokady jednostek wczytujących i zapisujących są implementowane za pomocą blokowania każdego obiektu z bazy. Blokady mogą działać w trybie *współużytkowanym* lub *wyłącznym*. Blokada jest używana w następujący sposób:

- Jeśli transakcja chce wczytać obiekt, musi najpierw zająć blokadę w trybie współużytkowanym. W tym trybie wiele transakcji może jednocześnie zajmować blokadę. Jednak gdy inna transakcja zajęła już blokadę w trybie wyłącznym, te transakcje muszą czekać.
- Gdy transakcja chce zapisać dane w obiekcie, musi najpierw zająć blokadę na wyłączność. Żadna inna transakcja nie może w tym samym czasie zajmować blokady (ani w trybie współużytkowanym, ani na wyłączność). Dlatego jeśli istnieje już blokada danego obiektu, transakcja musi czekać.
- Jeśli transakcja najpierw wczytuje, a następnie zapisuje obiekt, może zmienić blokadę współużytkowaną na wyłączną. Ta zmiana odbywa się tak samo jak bezpośrednie zajmowanie blokady wyłącznej.

- Gdy transakcja otrzyma blokadę, musi utrzymywać ją do momentu zakończenia transakcji (zatwierdzenia lub anulowania jej). Tego dotyczy człon „dwuetapowe” — w pierwszej fazie (w trakcie wykonywania transakcji) blokada jest zajmowana, a w drugiej (na zakończenie transakcji) wszystkie blokady są zwalniane.

Ponieważ używanych jest tak wiele blokad, łatwo może dojść do tego, że transakcja A utknie w oczekiwaniu na zwolnienie blokady przez transakcję B i na odwrót. Taka sytuacja to *zakleszczenie*. Baza automatycznie wykrywa zakleszczenie różnych transakcji i anuluje jedną z nich, aby inne mogły kontynuować pracę. Aplikacja musi ponowić anulowaną transakcję.

## Wydajność przy stosowaniu blokad dwuetapowych

Poważną wadą blokad dwuetapowych i powodem, dla którego nie są one powszechnie stosowane od lat 70., jest wydajność tej techniki. Przepustowość transakcji i czas odpowiedzi na zapytania są dla blokad dwuetapowych znacznie gorsze niż przy słabej izolacji.

Po części wynika to z kosztów zajmowania i zwalniania blokad. Ważniejsze jest jednak ograniczenie współbieżności. Z projektu wynika, że jeśli dwie współbieżne transakcje próbują zrobić coś, co może skutkować sytuacją wyścigu, jedna z nich musi oczekiwać na zakończenie pracy przez drugą.

Tradycyjne bazy relacyjne nie ograniczają czasu trwania transakcji, ponieważ są zaprojektowane pod kątem aplikacji interaktywnych, oczekujących na dane wejściowe od użytkownika. Dlatego gdy jedna transakcja musi oczekiwać na inną, czas oczekiwania jest nieograniczony. Nawet jeśli zadbasz o to, by wszystkie transakcje były krótkie, może powstać kolejka, gdy kilka transakcji zechce uzyskać dostęp do tego samego obiektu. Dlatego transakcja musi oczekiwać na zakończenie kilku innych, zanim będzie mogła cokolwiek zrobić.

Z tego powodu opóźnienia w bazach używających blokad dwuetapowych mogą być niestabilne. Gdy obciążenie robocze związane jest z dużą rywalizacją transakcji, na wyższych percentylach mogą one działać bardzo powoli (zob. punkt „Opis wydajności”). Może wystarczyć jedna powolna transakcja lub jedna transakcja używająca wielu danych i zajmująca liczne blokady, aby reszta systemu się zatrzymała. Taka niestabilność sprawia problemy, gdy wymagane jest niezawodne działanie systemu.

Choć przy izolacji opartej na blokadach izolacji na poziomie odczytu zatwierdzonych danych zakleszczenie może wystąpić, zdarza się ono znacznie częściej przy izolacji na poziomie sekwencyjności z blokadami dwuetapowymi (zależy od wzorca dostępu do danych w transakcjach). Może to stanowić dodatkowy problem w obszarze wydajności. Jeśli transakcja zostanie anulowana z powodu zakleszczenia, a następnie ponowiona, będzie musiała ponownie wykonać całą pracę. Gdy zakleszczenie występuje często, może powodować marnowanie dużej ilości pracy.

## Blokady oparte na predykatkach

W poprzednim opisie blokad pominięty został subtelny, ale ważny szczegół. W punkcie „Zapis zniekształcający i fantomy” omówiono problem *fantomów*, polegający na tym, że jedna transakcja zmienia wyniki zapytania z wyszukiwaniem w innej transakcji. Baza z izolacją na poziomie sekwencyjności musi zapobiegać fantomom.

W scenariuszu rezerwowania sali konferencyjnej oznacza to, że jeśli jedna transakcja szuka istniejących rezerwacji sali w określonym oknie czasowym (zob. listing 7.2), inna transakcja nie może jednocześnie wstawić ani zaktualizować innej rezerwacji tego samego pomieszczenia na uwzględniany czas. Dopuszczalne jest jednoczesne dodawanie rezerwacji innych sal lub tego samego pomieszczenia w innych godzinach, jeśli nie jest to sprzeczne z proponowaną rezerwacją.

Jak zaimplementować takie rozwiązanie? Potrzebna jest *blokada oparta na predykanie* [3]. Działa ona podobnie jak opisane wcześniej blokady współużytkowane i wyłączone, jednak dotyczy nie tyle konkretnego obiektu (np. wiersza tabeli), co wszystkich obiektów spełniających jakiś warunek wyszukiwania:

```
SELECT * FROM bookings
WHERE room_id = 123 AND
      end_time > '2018-01-01 12:00' AND
      start_time < '2018-01-01 13:00';
```

Blokada oparta na predykatkach ogranicza dostęp w następujący sposób:

- Jeśli transakcja A chce wczytać obiekty zgodne z jakimś warunkiem (tak jak w podanym zapytaniu SELECT), musi zająć opartą na predykanie blokadę współużytkowaną na podstawie warunków z zapytania. Gdy inna transakcja B zajmuje obecnie blokadę wyłączną któregośkolwiek obiektu pasującego do tych warunków, A musi poczekać na zwolnienie blokady przez B. Dopiero później może wykonać swoje zapytanie.
- Jeżeli transakcja A chce wstawić, zaktualizować lub usunąć dowolny obiekt, najpierw musi sprawdzić, czy dawna lub nowa wartość pasują do którejkolwiek z istniejących blokad opartych na predykanie. Gdy istnieje pasująca blokada oparta na predykanie zajmowana przez transakcję B, A musi poczekać na zatwierdzenie lub anulowanie transakcji B; dopiero potem może kontynuować pracę.

Główna idea polega tu na tym, że blokada oparta na predykanie dotyczy nawet obiektów nieistniejących w bazie, które jednak mogą zostać dodane w przyszłości (czyli fantomów). Jeśli blokada dwuetapowa obejmuje blokady oparte na predykanie, baza zapobiega wszelkim formom zapisu zniekształcającego i innym sytuacjom wyjścia. Uzyskiwana jest więc izolacja na poziomie sekwencyjności.

## Blokady zakresów indeksu

Niestety, wydajność blokad opartych na predykatkach jest niska. Jeśli aktywne transakcje utrzymują wiele blokad, sprawdzanie blokad zajmuje dużo czasu. Dlatego większość baz stosujących blokady dwuetapowe implementuje *blokady zakresów indeksu* (*blokady następnego klucza*), które są uproszczonym przybliżeniem blokad opartych na predykatkach [41, 50].

Można bezpiecznie uprościć predykat, uwzględniając w warunku większy zbiór obiektów. Na przykład jeśli blokada oparta na predykanie dotyczy rezerwacji sali 123 od południa do 13:00, możesz utworzyć jej przybliżoną wersję, blokując wszystkie sale (nie tylko 123) od południa do 13:00. Jest to bezpieczne, ponieważ każdy zapis pasujący do pierwotnego predykatu z pewnością pasuje też do przybliżenia.

W bazie z rezerwacjami sal prawdopodobnie znajduje się indeks dla kolumny `room_id` i/lub indeksy dla kolumn `start_time` oraz `end_time` (w przeciwnym razie pokazane zapytanie w dużej bazie byłoby bardzo powolne):

- Załóżmy, że istnieje indeks dla kolumny `room_id`, a baza używa go do znalezienia istniejących rezerwacji sali 123. Wtedy baza może dodać blokadę współużytkowaną dla danej pozycji indeksu, oznaczającą, że transakcja szukała rezerwacji sali 123.
- Inna możliwość jest taka, że baza szuka istniejących rezerwacji za pomocą indeksu opartego na czasie. Wtedy może dodać blokadę współużytkowaną dla zakresu wartości z tego indeksu. Oznacza to, że transakcja wyszukiwała rezerwacje z czasu od południa do 13:00 1 stycznia 2018.

W obu scenariuszach z jednym z indeksów wiązane jest przybliżenie warunku z wyszukiwania. Jeśli teraz inna transakcja zechce wstawić, zaktualizować lub usunąć rezerwację tej samej sali i/lub rezerwację w określonym czasie, będzie musiała zaktualizować tę samą część indeksu. W tym procesie natrafi na blokadę współużytkowaną i będzie musiała poczekać na jej zwolnienie.

Stanowi to skuteczną ochronę przed fantomami i zapisem zniekształcającym. Blokadę zakresów indeksu nie są tak precyzyjne jak blokady oparte na predykcji (mogą blokować większy zakres obiektów niż jest to konieczne do zachowania sekwencyjności). Jednak ponieważ powodują znacznie mniejsze koszty, stanowią dobry kompromis.

Jeśli nie istnieje indeks, do którego można dodać blokadę zakresu, baza może zastosować blokadę współużytkowaną dotyczącą całej tabeli. Nie jest to korzystne ze względu na wydajność, ponieważ uniemożliwia zapis w tabeli wszystkim innym transakcjom, jest to jednak bezpieczne rozwiązanie rezerwowe.

## Algorytm SSI

W tym rozwiązaniu zarysowano ponurą wizję zarządzania współbieżnością w bazach. Z jednej strony istnieją implementacje sekwencyjności, które mają niską wydajność (blokady dwuetapowe) lub słabo się skalują (wykonywanie sekwencyjne). Z drugiej strony występują niskie poziomy izolacji o dobrej wydajności, ale narażone na różne sytuacje wyścigu (utrata aktualizacji, zapis zniekształcający, fantomy itd.). Czy izolacja na poziomie sekwencyjności i wysoka wydajność zasadniczo się wykluczają?

Możliwe, że nie. Bardzo obiecujący jest algorytm **SSI** (ang. *serializable snapshot isolation*, czyli izolacja snapshotów z zapewnianiem sekwencyjności). Zapewnia on pełną sekwencyjność, a przy tym powoduje tylko niewielki spadek wydajności w porównaniu z izolacją snapshotów. SSI to stosunkowo nowy algorytm. Został opisany po raz pierwszy w 2008 r. [40] i jest tematem pracy doktorskiej Michaela Cahilla [51].

Dziś SSI jest używany zarówno w bazach działających w jednym węźle (izolacja na poziomie sekwencyjności w bazie PostgreSQL od wersji 9.1 [41]), jak i w bazach rozproszonych (w bazie FoundationDB używany jest podobny algorytm). Algorytm SSI jest na tyle nowy w porównaniu z innymi mechanizmami zarządzania współbieżnością, że wciąż jeszcze musi udowodnić wysoką wydajność w praktyce. Jest jednak możliwe, że okaże się tak szybki, iż w przyszłości stanie się nowym domyślnym rozwiązaniem.

## Pesymistyczne i optymistyczne zarządzanie współbieżnością

Blokady dwuetapowe to tzw. *pesymistyczny* mechanizm zarządzania współbieżnością. Jest on oparty na regule, że jeśli coś może się nie powieść (na co wskazuje blokada zajmowana przez inną transakcję), przed wykonaniem jakichkolwiek operacji lepiej jest poczekać do czasu, aż sytuacja stanie się bezpieczna. Ten mechanizm działa jak *wzajemne wykluczanie* służące do ochrony struktur danych w programowaniu wielowątkowym.

Wykonywanie sekwencyjne jest, w pewnym sensie, skrajnie pesymistyczne. Działa podobnie jak zajęcie przez każdą transakcję blokady wyłącznej całej bazy (lub jednej partycji) na czas wykonywania transakcji. Ten pesymizm jest kompensowany tworzeniem bardzo szybkich transakcji, dzięki czemu „blokadę” trzeba zajmować tylko na krótki czas.

Z kolei algorytm SSI to *optymistyczna* technika zarządzania współbieżnością. „Optymistyczna” oznacza w tym kontekście tyle, że zamiast tworzyć blokadę, gdy może się stać coś potencjalnie niebezpiecznego, transakcja kontynuuje pracę w nadziei, iż wszystko potoczy się dobrze. Gdy transakcja chce zatwierdzić operację, baza sprawdza, czy stało się coś złego (czy izolacja została naruszona). Wystąpienie problemów powoduje, że transakcja jest anulowana i trzeba ją ponowić. Tylko transakcje, które zostały wykonane sekwencyjnie, mogą zostać zatwierdzone.

Optymistyczne zarządzanie współbieżnością to stary pomysł [52]. Nad wadami i zaletami tej techniki dyskutowano przez długi czas [53]. Jej wydajność jest niska, gdy rywalizacja jest wysoka (wiele transakcji próbuje uzyskać dostęp do tych samych obiektów), ponieważ skutkuje to tym, że duży odsetek transakcji trzeba anulować. Jeśli system zbliża się do maksymalnej przepustowości, dodatkowe obciążenie spowodowane ponawianymi transakcjami może pogorszyć wydajność.

Jednak gdy system ma wystarczającą ilość wolnych zasobów, a rywalizacja między transakcjami nie jest wysoka, optymistyczne techniki zarządzania współbieżnością zwykle są wydajniejsze od pesymistycznych. Rywalizację można ograniczyć dzięki przemianom operacjom atomowym. Na przykład gdy kilka transakcji chce jednocześnie zwiększyć wartość licznika, nie ma znaczenia, w jakiej kolejności to zrobią (o ile licznik nie jest wczytywany w tej samej transakcji). Dlatego wszystkie współbieżne inkrementacje można wprowadzić bez powodowania konfliktu.

Jak wskazuje na to nazwa, algorytm SSI jest oparty na izolacji snapshotów. Oznacza to, że wszystkie odczyty w transakcji są wykonywane z użyciem spójnego snapshota bazy (zob. punkt „Izolacja snapshotów i powtarzalny odczyt”). Jest to główna różnica w porównaniu z wcześniejszymi optymistycznymi technikami zarządzania współbieżnością. Obok izolacji snapshotów SSI obejmuje algorytm wykrywania konfliktów w sekwencji zapisów i ustalania, które transakcje należy anulować.

## Decyzje oparte na nieaktualnych założeniach

We wcześniejszym opisie zapisu zniekształcającego w izolacji snapshotów (zob. punkt „Zapis zniekształcający i fantomy”) wystąpił powtarzający się wzorec: transakcja wczytuje dane z bazy, sprawdza wynik zapytania i decyduje się wykonać jakieś operacje (zapis w bazie) na podstawie otrzymanego wyniku. Jednak w izolacji snapshotów wynik z pierwotnego zapytania może stać się nieaktualny do momentu zatwierdzenia transakcji, ponieważ dane mogły w międzyczasie zostać zmodyfikowane.

Można ująć to inaczej: transakcja podejmuje działania na podstawie *założenia* (faktu, który był prawdziwy na początku transakcji, np. „Obecnie na dyżurze jest dwóch lekarzy”). Później gdy transakcja chce zatwierdzić operację, pierwotne dane mogły się zmienić i założenie niekoniecznie jest dalej prawdziwe.

Gdy aplikacja zgłasza zapytanie (np. „Ilu lekarzy jest obecnie na dyżurze?”), baza nie wie, w jaki sposób logika aplikacji wykorzystuje wynik tego zapytania. Aby zapewnić bezpieczeństwo, baza musi zakładać, że wszelkie zmiany w wyniku zapytania (założeniu) oznaczają, iż transakcja może być nieprawidłowa. Oznacza to, że może występować zależność przyczynowa między zapytaniami a zapisami transakcji. Aby zapewnić izolację na poziomie sekwencyjności, baza musi wykrywać sytuacje, w których transakcja działała na podstawie nieaktualnych założeń, i anulować takie transakcje.

W jaki sposób baza wie, że wynik zapytania mógł się zmienić? Trzeba uwzględnić dwie sytuacje:

- Wykrywać odczyty nieaktualnych wersji obiektu w technice MVCC (niezatwierdzony zapis nastąpił przed odczytem).
- Wykrywać zapisy wpływające na wcześniejsze odczyty (zapis nastąpił po odczycie).

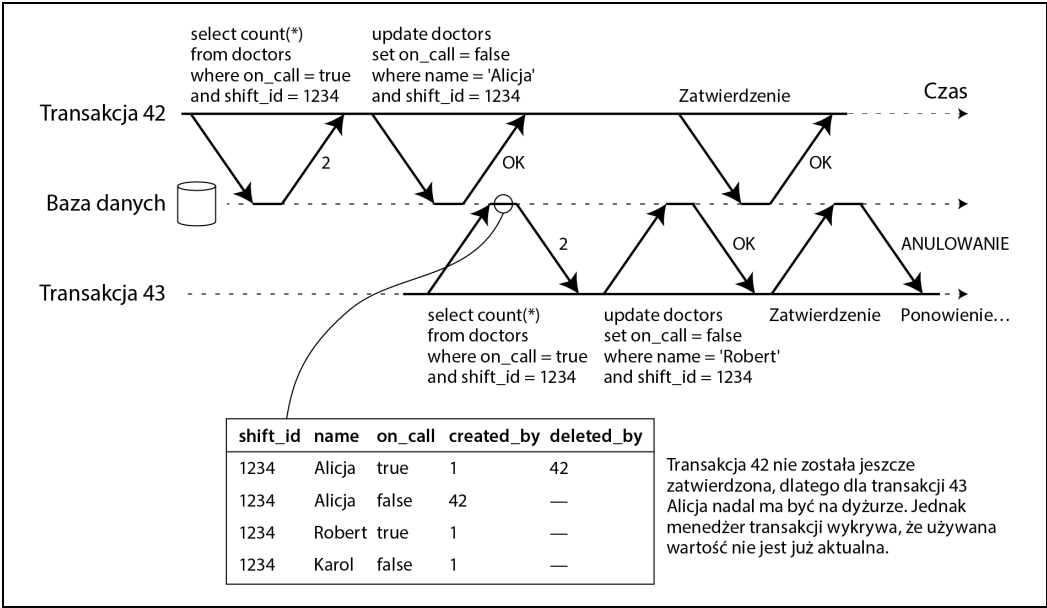
### Wykrywanie odczytu nieaktualnych wersji w technice MVCC

Przypomnij sobie, że izolacja snapshotów jest zwykle implementowana za pomocą techniki MVCC (zob. rysunek 7.10). Gdy transakcja wczytuje dane ze spójnego snapshota w bazie używającej techniki MVCC, ignoruje zapisy wprowadzone przez inne transakcje, które w momencie wykonania snapshota nie zostały jeszcze zatwierdzone. Na rysunku 7.10 transakcja 43 widzi dotyczącą Alicji wartość `on_call = true`, ponieważ transakcja 42 (która zmodyfikowała status dyżuru Alicji) nie jest zatwierdzona. Jednak gdy transakcja 43 chce zatwierdzić operację, transakcja 42 została już zatwierdzona. To oznacza, że zapis zignorowany w trakcie odczytu danych ze spójnego snapshota, został wprowadzony, dlatego założenie z transakcji 43 nie jest już prawdziwe.

Aby zapobiec tej anomalii, baza musi wykrywać, że transakcja ignoruje zapisy innej transakcji z powodu reguł widoczności z techniki MVCC. Gdy transakcja chce zatwierdzić operację, baza sprawdza, czy któryś ze zignorowanych zapisów nie został zatwierdzony. Zatwierdzenie zapisu wymaga anulowania transakcji.

Po co czekać do momentu zatwierdzenia operacji? Dlaczego nie anulować transakcji 43 natychmiast po wykryciu odczytu nieaktualnych danych? No cóż, gdyby transakcja 43 jedynie wczytywała dane, jej anulowanie nie byłoby konieczne, ponieważ nie byłoby ryzyka zapisu zniekształcającego. W momencie wczytywania danych przez transakcję 43 baza jeszcze nie wie, czy transakcja później wykona zapis. Ponadto w momencie zatwierdzania transakcji 43 transakcja 42 może być już anulowana lub wciąż niezatwierdzona. Dlatego może się okazać, że odczyt nie jest przestarzały. Dzięki unikaniu niepotrzebnego anulowania transakcji technika SSI zapewnia obsługę izolacji snapshotów dla długich odczytów ze spójnego snapshota.

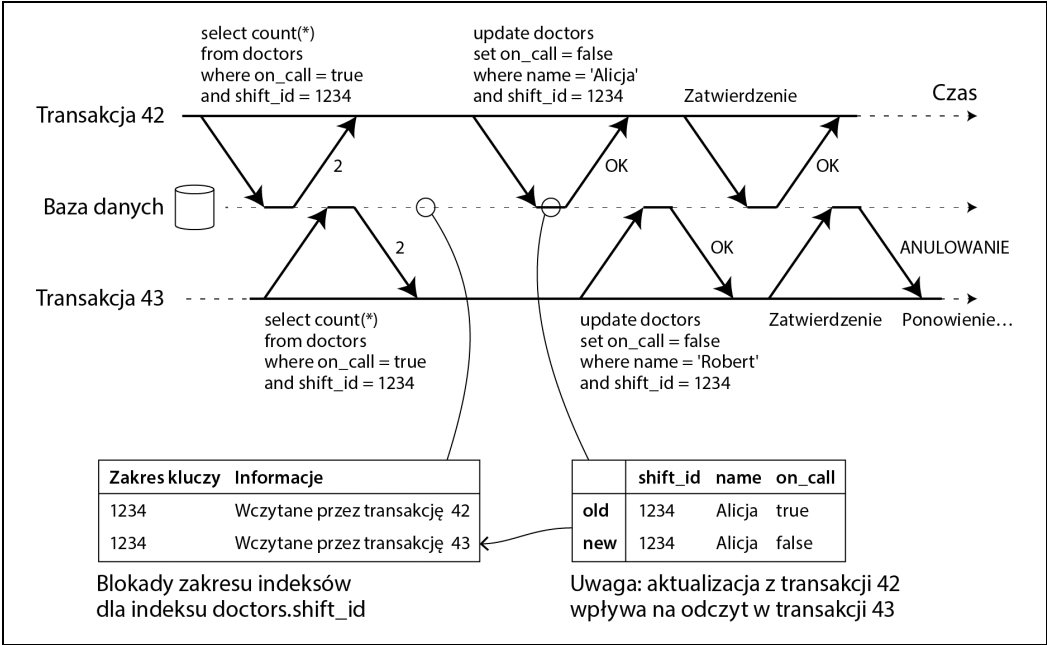




Rysunek 7.10. Wykrywanie sytuacji, gdy transakcja wczytuje nieaktualne wartości ze snapshota w technice MVCC

### Wykrywanie zapisów wpływających na wcześniejsze odczyty

Drugi scenariusz dotyczy modyfikowania danych przez transakcję po ich wczytaniu. Tę sytuację przedstawia rysunek 7.11.



Rysunek 7.11. Algorytm SSI — wykrywanie sytuacji, gdy jedna transakcja modyfikuje odczyty innej transakcji

W kontekście blokad dwuetapowych opisano blokady zakresów indeksów (zob. punkt „Blokady zakresów indeksu”). Umożliwiają one bazie zablokowanie dostępu do wszystkich wierszy pasujących do określonego zapytania, np. `WHERE shift_id = 1234`. Tu można zastosować podobną technikę, przy czym blokady w SSI nie blokują innych transakcji.

Na rysunku 7.11 obie transakcje, 42 i 43, szukają dyżurnych lekarzy ze zmiany 1234. Jeśli istnieje indeks dla kolumny `shift_id`, baza może wykorzystać indeks 1234 do zarejestrowania tego, że transakcje 42 i 43 wczytały te dane. Jeżeli indeks nie istnieje, tę informację można zapisać na poziomie tabeli. Taką informację trzeba przechowywać tylko przez pewien czas. Po zakończeniu (zatwierdzeniu lub anulowaniu) danej transakcji i wszystkich jednocześnie wykonywanych transakcji baza może „zapomnieć”, jakie dane te operacje wczytywały.

Gdy transakcja zapisuje dane w bazie, musi sprawdzić w indeksie wszystkie inne transakcje, które niedawno wczytały modyfikowane dane. Ten proces przypomina zajmowanie blokady zapisu dotyczącej modyfikowanego zakresu kluczy. Jednak zamiast blokować dane do czasu zatwierdzenia operacji przez jednostki wczytujące, blokada działa jak ostrzeżenie i jedynie powiadamia transakcje, że wczytane dane mogą być nieaktualne.

Na rysunku 7.11 transakcja 43 powiadamia transakcję 42, że jej wcześniejszy odczyt jest nieaktualny (i na odwrót). Transakcja 42 jako pierwsza zatwierdza operacje i robi to z powodzeniem. Choć zapis z transakcji 43 miał wpływ na transakcję 42, transakcja 43 nie została jeszcze zatwierdzona, dlatego wykonywany przez nią zapis na razie nie wszedł w życie. Jednak gdy transakcja 43 chce zatwierdzić zmiany, sprzeczny zapis z transakcji 42 został już zatwierdzony. Z tego powodu transakcję 43 trzeba anulować.

## Wydajność techniki SSI

Na działanie algorytmu w praktyce jak zawsze wpływa wiele szczegółów inżynierskich. Na przykład jeden z kompromisów dotyczy precyzji śledzenia odczytów i zapisów z transakcji. Jeśli baza bardzo szczegółowo śledzi aktywność każdej transakcji, potrafi precyzyjnie określić, które transakcje trzeba anulować. Jednak koszty obsługi rozwiązania mogą być wtedy wysokie. Mniej szczegółowe śledzenie jest szybsze, jednak może prowadzić do anulowania większej liczby transakcji, niż jest to konieczne.

W niektórych sytuacjach dopuszczalne jest, by transakcja wczytywała informacje nadpisane przez inną transakcję. W zależności od tego, co jeszcze się stało, czasem można udowodnić, że transakcje i tak działały zgodnie z sekwencyjnością. W bazie PostgreSQL ta teoria jest wykorzystywana do zmniejszenia liczby niepotrzebnych przypadków anulowania transakcji [11, 41].

W porównaniu z blokadami dwuetapowymi ważną zaletą techniki SSI jest to, że nie trzeba wstrzymywać pracy transakcji w oczekiwaniu na blokady zajmowane przez inną transakcję. Tak jak w izolacji snapshotów jednostki zapisujące nie blokują jednostek wczytujących i na odwrót. Ta reguła projektowa sprawia, że czas przetwarzania zapytań jest dużo bardziej przewidywalny i mniej zmienny. Ważne jest przede wszystkim to, że zapytania tylko wczytujące dane można uruchamiać z użyciem spójnego snapshota bez konieczności zajmowania blokad. Jest to bardzo przydatne w obciążeniu roboczym z dużą liczbą odczytów.

W porównaniu z sekwencyjnym wykonywaniem kodu technika SSI nie jest ograniczona do przepustowości jednego rdzenia procesora. Baza FoundationDB rozdziela wykrywanie konfliktów związanych z sekwencyjnością między wiele maszyn, co pozwala skalować rozwiązanie pod kątem bardzo wysokiej przepustowości. Choć dane mogą zostać podzielone między wiele maszyn, transakcje mogą czytać i zapisywać dane w wielu partycjach przy zachowaniu izolacji na poziomie sekwencyjności [54].

Liczba przypadków anulowania ma duży wpływ na ogólną wydajność techniki SSI. Na przykład transakcja, która wczytuje i zapisuje dane przez długi czas, zapewne natrafi na konflikt i będzie wymagała anulowania. Dlatego SSI wymaga, by transakcje z odczytem i zapisem były stosunkowo krótkie (długie transakcje z samym odczytem mogą być akceptowalne). Jednak technika SSI jest prawdopodobnie mniej wrażliwa na powolne transakcje niż blokady dwuetapowe lub sekwencyjne wykonywanie transakcji.

## Podsumowanie

Transakcje to warstwa abstrakcji umożliwiająca aplikacji udawanie, że niektóre problemy ze współbieżnością oraz pewne błędy sprzętowe i programowe nie istnieją. Duża klasa błędów jest zredukowana do prostego *anulowania transakcji*, po czym aplikacja musi jedynie ponowić próbę.

W tym rozdziale pokazano wiele przykładowych problemów, którym transakcje mogą pomagać zapobiegać. Nie wszystkie aplikacje są podatne na wszystkie takie problemy. W aplikacji z bardzo prostymi wzorcami dostępu, np. wczytującej i zapisującej tylko pojedyncze rekordy, prawdopodobnie można się obejść bez transakcji. Jednak przy bardziej złożonych wzorcach dostępu transakcje mogą znacznie zmniejszyć liczbę potencjalnych rodzajów błędów, które trzeba przemyśleć.

Bez transakcji różne rodzaje błędów (awarie procesów, zakłócenia w pracy sieci, brak zasilania, przepełnienie dysku, nieoczekiwana współbieżność itd.) powodują, że dane mogą się stać niespójne na różne sposoby. Na przykład zdenormalizowane dane są w stanie łatwo utracić synchronizację z danymi źródłowymi. Bez transakcji bardzo trudno jest analizować wpływ złożonych i powiązanych ze sobą dostępów do danych na bazę.

W tym rozdziale bardzo szczegółowo omówiono zagadnienie zarządzania współbieżnością. Opisano tu kilka często stosowanych poziomów izolacji, a zwłaszcza *odczyt zatwierdzonych danych*, *izolację snapshotów* (nazywaną czasem *powtarzalnym odczytem*) i *sekwencyjność*. Charakterystykę tych poziomów izolacji przedstawiono na podstawie różnych przykładów sytuacji wyjściu:

### *Brudne odczyty*

Jeden klient wczytuje zapisy innego klienta przed ich zatwierdzeniem. Zapobiegają temu odczyt zatwierdzonych danych i wyższe poziomy izolacji.

### *Brudne zapisy*

Jeden klient nadpisuje dane zapisane, ale niezatwierdzone jeszcze przez innego. Prawie wszystkie implementacje transakcji zapobiegają brudnym zapisom.

### *Zapis zniekształcający (brak powtarzalności odczytów)*

Klient widzi różne części bazy z różnych momentów w czasie. Temu problemowi najczęściej się zapobiega za pomocą izolacji snapshotów, dzięki czemu transakcja może wczytywać dane ze spójnego snapshota z określonego momentu w czasie. To podejście jest zwykle implementowane za pomocą techniki MVCC.

### *Utrata aktualizacji*

Dwa klienty jednocześnie wykonują cykl wczytaj – zmodyfikuj – zapisz. Jeden klient nadpisuje zapis drugiego bez uwzględniania wprowadzonych przez tego ostatniego zmian, dlatego dane zostają utracone. Niektóre implementacje izolacji snapshotów automatycznie zapobiegają tej anomalii, natomiast inne wymagają ręcznie dodawanych blokad (SELECT FOR UPDATE).

### *Zapis zniekształcający*

Transakcja wczytuje dane, podejmuje decyzję na podstawie uzyskanej wartości, a następnie zapisuje tę decyzję w bazie. Jednak do czasu dokonania zapisu założenie stojące u podstaw tej decyzji staje się nieprawdziwe. Tej anomalii zapobiega tylko izolacja na poziomie sekwencyjności.

### *Odczyt fantomów*

Transakcja wczytuje obiekty pasujące do określonego warunku wyszukiwania. Inny klient dokonuje zapisu, który wpływa na wyniki tego wyszukiwania. Izolacja snapshotów zapobiega prostym odczytom fantomów, jednak fantomy w kontekście zapisu zniekształcającego wymagają specjalnego traktowania, np. blokad zakresów indeksów.

Niskie poziomy izolacji chronią przed niektórymi z tych anomalii, jednak wymagają od programisty aplikacji ręcznego radzenia sobie z innymi problemami (np. za pomocą bezpośrednio dodawanych blokad). Tylko izolacja na poziomie sekwencyjności chroni przed wszystkimi wymienionymi kłopotami. Omówiono tu trzy różne sposoby implementowania transakcji z zachowaniem sekwencyjności:

### *Wykonywanie transakcji dosłownie w porządku sekwencyjnym*

Jeśli możesz sprawić, by transakcje były wykonywane bardzo szybko, a liczba transakcji jest na tyle niska, by można je było przetwarzać w jednym rdzeniu procesora, jest to proste i skuteczne rozwiązanie.

### *Blokady dwuetapowe*

Przez dziesięciolecia był to standardowy sposób implementowania sekwencyjności. Jednak w wielu aplikacjach unika się go z powodu charakterystyki wydajności.

### *SSI*

Jest to stosunkowo nowy algorytm wolny od większości wad wcześniejszych podejść. Używane jest w nim podejście optymistyczne, co pozwala wykonywać transakcje bez blokowania ich. Gdy transakcja chce zatwierdzić operację, jest sprawdzana, a jeśli jej działanie nie było sekwencyjne, zostaje anulowana.

W przykładach z tego rozdziału używany jest relacyjny model danych. Jednak, co opisano w punkcie „Potrzeba stosowania transakcji dla wielu obiektów”, transakcje są wartościową funkcją baz niezależnie od używanego modelu.

W tym rozdziale przedstawiono pomysły i algorytmy głównie w kontekście baz działających w jednej maszynie. Transakcje w bazach rozproszonych otwierają nowy zestaw trudnych wyzwań opisanych w dwóch następnych rozdziałach.

### Literatura cytowana

- [1] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen i in., *A History and Evaluation of System R*, „Communications of the ACM”, rocznik 24, nr 10, s. 632 – 646, październik 1981 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.348&rep=rep1&type=pdf>; <https://dl.acm.org/citation.cfm?doid=358769.358784>).
- [2] Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu i Irving L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, w: „Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems”, red. Gerardus Maria Nijssen, s. 364 – 394, Elsevier/North Holland Publishing, 1976 (<http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8248&rep=rep1&type=pdf>); także w: *Readings in Database Systems*, wydanie czwarte, red. Joseph M. Hellerstein i Michael Stonebraker, MIT Press, 2005, ISBN: 978-0-262-69314-1.
- [3] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie i Irving L. Traiger, *The Notions of Consistency and Predicate Locks in a Database System*, „Communications of the ACM”, rocznik 19, nr 11, s. 624 – 633, listopad 1976 (<http://jimgray.azurewebsites.net/papers/on%20the%20notions%20of%20consistency%20and%20predicate%20locks%20in%20a%20database%20system%20cacm.pdf>).
- [4] *ACID Transactions Are Incredibly Helpful*, FoundationDB, LLC, 2013 (<http://web.archive.org/web/20150320053809/https://foundationdb.com/acid-claims>).
- [5] John D. Cook, *ACID Versus BASE for Database Transactions*, [johndcook.com](http://johndcook.com), 6 czerwca 2009 (<https://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/>).
- [6] Gavin Clarke, *NoSQL’s CAP Theorem Busters: We Don’t Drop ACID*, [theregister.co.uk](http://theregister.co.uk), 22 listopada 2012 ([http://www.theregister.co.uk/2012/11/22/foundationdb\\_fear\\_of\\_cap\\_theorem/](http://www.theregister.co.uk/2012/11/22/foundationdb_fear_of_cap_theorem/)).
- [7] Theo Härder i Andreas Reuter, *Principles of Transaction-Oriented Database Recovery*, „ACM Computing Surveys”, rocznik 15, nr 4, s. 287 – 317, grudzień 1983 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2812&rep=rep1&type=pdf>; <https://dl.acm.org/citation.cfm?doid=289.291>).
- [8] Peter Bailis, Alan Fekete, Ali Ghodsi i in., *HAT, not CAP: Towards Highly Available Transactions*, w: „14th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2013 (<http://www.bailis.org/papers/hat-hotos2013.pdf>).
- [9] Armando Fox, Steven D. Gribble, Yatin Chawathe i in., *Cluster-Based Scalable Network Services*, w: „16th ACM Symposium on Operating Systems Principles” (SOSP), październik 1997 (<https://people.eecs.berkeley.edu/~brewer/cs262b/TACC.pdf>).

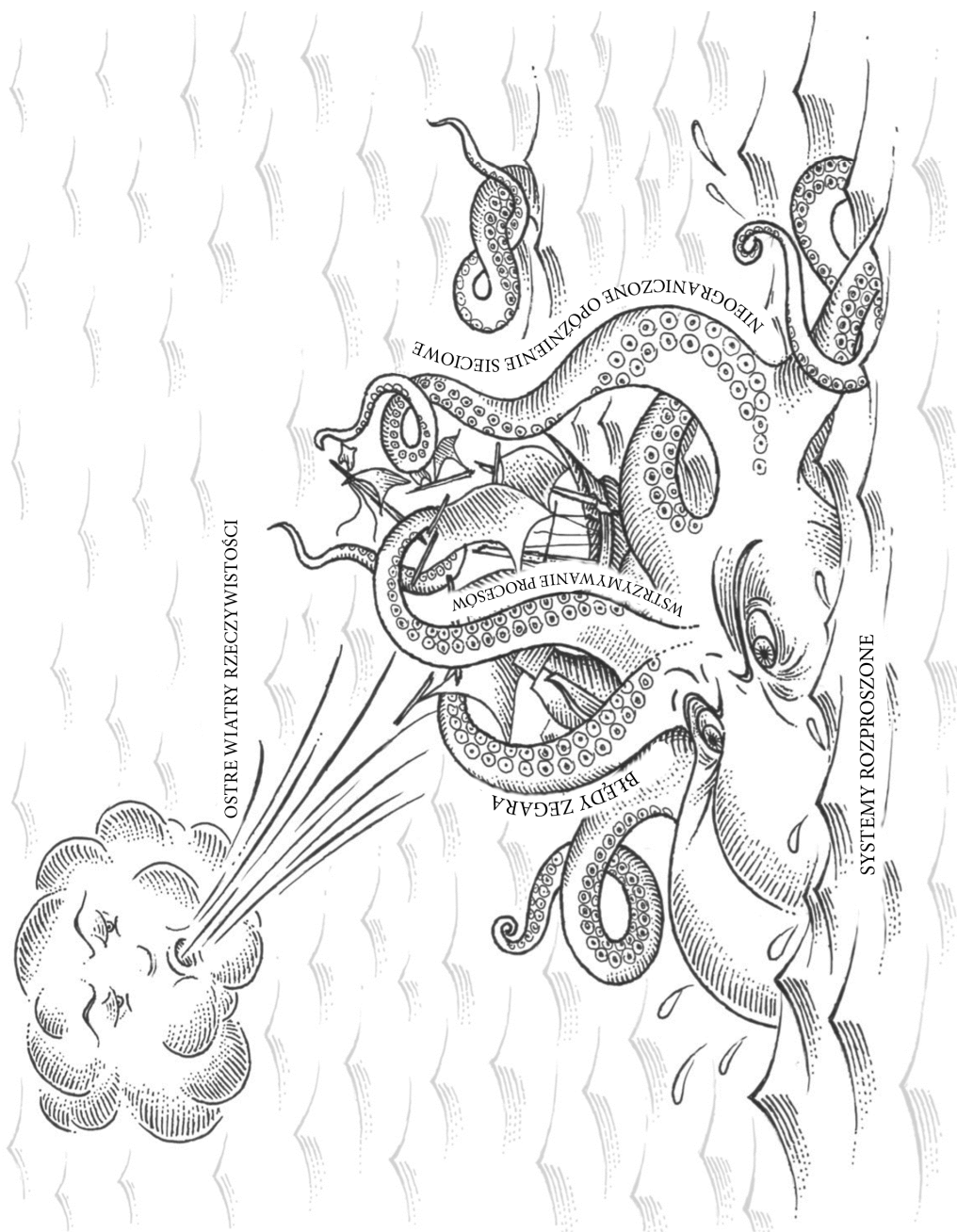
- [10] Philip A. Bernstein, Vassos Hadzilacos i Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987, ISBN: 978-0-201-10715-9; dostępne w internecie w serwisie [research.microsoft.com](https://www.microsoft.com/en-us/research/people/philbe/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fpeople%2Fphilbe%2Fcontrol.aspx) (<https://www.microsoft.com/en-us/research/people/philbe/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fpeople%2Fphilbe%2Fcontrol.aspx>).
- [11] Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil i in., *Making Snapshot Isolation Serializable*, „ACM Transactions on Database Systems”, rocznik 30, nr 2, s. 492 – 528, czerwiec 2005 (<https://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/2009/Papers/p492-fekete.pdf>; <https://dl.acm.org/citation.cfm?doid=1071610.1071615>).
- [12] Mai Zheng, Joseph Tucek, Feng Qin i Mark Lillibridge, *Understanding the Robustness of SSDs Under Power Fault*, w: „11th USENIX Conference on File and Storage Technologies” (FAST), luty 2013 (<https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf>).
- [13] Laurie Denness, *SSDs: A Gift and a Curse*, [laur.ie](http://laur.ie), 2 czerwca 2015 (<https://laur.ie/blog/2015/06/ssds-a-gift-and-a-curse/>).
- [14] Adam Surak, *When Solid State Drives Are Not That Solid*, [blog.algolia.com](http://blog.algolia.com), 15 czerwca 2015 (<https://blog.algolia.com/when-solid-state-drives-are-not-that-solid/>).
- [15] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnathan Alagappan i in., *All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications*, w: „11th USENIX Symposium on Operating Systems Design and Implementation” (OSDI), październik 2014 (<http://research.cs.wisc.edu/wind/Publications/alice-osdi14.pdf>).
- [16] Chris Siebenmann, *Unix’s File Durability Problem*, [utcc.utoronto.ca](http://utcc.utoronto.ca), 14 kwietnia 2016 (<https://utcc.utoronto.ca/~cks/space/blog/unix/FileSyncProblem>).
- [17] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder i in., *An Analysis of Data Corruption in the Storage Stack*, w: „6th USENIX Conference on File and Storage Technologies” (FAST), luty 2008 (<http://research.cs.wisc.edu/adsl/Publications/corruption-fast08.pdf>).
- [18] Bianca Schroeder, Raghav Lagisetty i Arif Merchant, *Flash Reliability in Production: The Expected and the Unexpected*, w: „14th USENIX Conference on File and Storage Technologies” (FAST), luty 2016 (<https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder>).
- [19] Don Allison, *SSD Storage — Ignorance of Technology Is No Excuse*, [blog.korelogic.com](http://blog.korelogic.com), 24 marca 2015 (<https://blog.korelogic.com/blog/2015/03/24>).
- [20] Dave Scherer, *Those Are Not Transactions (Cassandra 2.0)*, [blog.foundationdb.com](http://blog.foundationdb.com), 6 września 2013 (<http://web.archive.org/web/20150526065247/http://blog.foundationdb.com/those-are-not-transactions-cassandra-2-0>).
- [21] Kyle Kingsbury, *Call Me Maybe: Cassandra*, [aphyr.com](http://aphyr.com), 24 września 2013 (<https://aphyr.com/posts/294-call-me-maybe-cassandra/>).
- [22] *ACID Support in Aerospike*, Aerospike, Inc., czerwiec 2014 (<https://www.aerospike.com/docs/architecture/assets/AerospikeACIDSupport.pdf>).

- [23] Martin Kleppmann, *Hermitage: Testing the 'I' in ACID*, [martin.kleppmann.com](http://martin.kleppmann.com), 25 listopada 2014 (<http://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html>).
- [24] Tristan D'Agosta, *BTC Stolen from Poloniex*, [bitcointalk.org](http://bitcointalk.org), 4 marca 2014 (<https://bitcointalk.org/index.php?topic=499580>).
- [25] bitcointhief2, „How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!”, [reddit.com](http://reddit.com), 2 lutego 2014 ([https://www.reddit.com/r/Bitcoin/comments/1wtbiu/how\\_i\\_stole\\_roughly\\_100\\_btc\\_from\\_an\\_exchange\\_and/](https://www.reddit.com/r/Bitcoin/comments/1wtbiu/how_i_stole_roughly_100_btc_from_an_exchange_and/)).
- [26] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham i S. Sudarshan, *Automating the Detection of Snapshot Isolation Anomalies*, w: „33rd International Conference on Very Large Data Bases” (VLDB), wrzesień 2007 (<http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf>).
- [27] Michael Melanson, *Transactions: The Limits of Isolation*, [michaelmelanson.net](http://michaelmelanson.net), 20 marca 2014 (<http://www.michaelmelanson.net/2014/12/01/transactions/>).
- [28] Hal Berenson, Philip A. Bernstein, Jim N. Gray i in., *A Critique of ANSI SQL Isolation Levels*, w: „ACM International Conference on Management of Data” (SIGMOD), maj 1995 (<https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F69541%2Ftr-95-51.pdf>).
- [29] Atul Adya, *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*, praca doktorska, Massachusetts Institute of Technology, marzec 1999 (<http://pmg.csail.mit.edu/papers/adya-phd.pdf>).
- [30] Peter Bailis, Aaron Davidson, Alan Fekete i in., *Highly Available Transactions: Virtues and Limitations (Extended Version)*, w: „40th International Conference on Very Large Data Bases” (VLDB), wrzesień 2014 (<https://arxiv.org/pdf/1302.0309.pdf>).
- [31] Bruce Momjian, *MVCC Unmasked*, [momjian.us](http://momjian.us), lipiec 2014 (<http://momjian.us/main/presentations/internals.html#mvcc>).
- [32] Annamalai Gurusami, *Repeatable Read Isolation Level in InnoDB — How Consistent Read View Works*, [blogs.oracle.com](http://blogs.oracle.com), 15 stycznia 2013 ([https://blogs.oracle.com/mysqlinnodb/entry/repeatable\\_read\\_isolation\\_level\\_in](https://blogs.oracle.com/mysqlinnodb/entry/repeatable_read_isolation_level_in)).
- [33] Nikita Prokopov, *Unofficial Guide to Datomic Internals*, [tonsky.me](http://tonsky.me), 6 maja 2014 (<http://tonsky.me/blog/unofficial-guide-to-datomic-internals/>).
- [34] Baron Schwartz, *Immutability, MVCC, and Garbage Collection*, [xaprb.com](http://xaprb.com), 28 grudnia 2013 (<https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/>).
- [35] J. Chris Anderson, Jan Lehnardt i Noah Slater, *CouchDB: The Definitive Guide*, O'Reilly Media, 2010, ISBN: 978-0-596-15589-6.
- [36] Rikdeb Mukherjee, *Isolation in DB2 (Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read) with Examples*, [mframes.blogspot.co.uk](http://mframes.blogspot.co.uk), 4 lipca 2013 (<http://mframes.blogspot.co.uk/2013/07/isolation-in-cursor.html>).

- [37] Steve Hilker, *Cursor Stability (CS)* — IBM DB2 Community, [toadworld.com](http://www.toadworld.com/platforms/ibmdb2/w/wiki/6661.cursor-stability-cs), 14 marca 2013 (<http://www.toadworld.com/platforms/ibmdb2/w/wiki/6661.cursor-stability-cs>).
- [38] Nate Wiger, *An Atomic Rant*, [nateware.com](http://www.nateware.com), 18 lutego 2010 (<http://www.nateware.com/an-atomic-rant.html#.Wjvyh1Xiat9>).
- [39] Joel Jacobson, *Riak 2.0: Data Types*, [blog.joeljacobsen.com](http://blog.joeljacobsen.com), 23 marca 2014.
- [40] Michael J. Cahill, Uwe Röhm i Alan Fekete, *Serializable Isolation for Snapshot Databases*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2008 (<https://cs.nyu.edu/courses/fall12/CSCI-GA.2434-001/p729-cahill.pdf>; <https://dl.acm.org/citation.cfm?doid=1376616.1376690>).
- [41] Dan R.K. Ports i Kevin Grittner, *Serializable Snapshot Isolation in PostgreSQL*, w: „38th International Conference on Very Large Databases” (VLDB), sierpień 2012 (<https://drkp.net/papers/ssi-vldb12.pdf>).
- [42] Tony Andrews, *Enforcing Complex Constraints in Oracle*, [tonyandrews.blogspot.co.uk](http://tonyandrews.blogspot.co.uk), 15 października 2004 (<http://tonyandrews.blogspot.co.uk/2004/10/enforcing-complex-constraints-in.html>).
- [43] Douglas B. Terry, Marvin M. Theimer, Karin Petersen i in., *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, w: „15th ACM Symposium on Operating Systems Principles” (SOSP), grudzień 1995 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.7889&rep=rep1&type=pdf>; <https://dl.acm.org/citation.cfm?doid=224056.224070>).
- [44] Gary Fredericks, *Postgres Serializability Bug*, [github.com](https://github.com), wrzesień 2015 (<https://github.com/gfredericks/pg-serializability-bug>).
- [45] Michael Stonebraker, Samuel Madden, Daniel J. Abadi i in., *The End of an Architectural Era (It's Time for a Complete Rewrite)*, w: „33rd International Conference on Very Large Data Bases” (VLDB), wrzesień 2007 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3697&rep=rep1&type=pdf>).
- [46] John Hugg, *H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures*, w: „Data @Scale Boston”, listopad 2014 (<https://www.youtube.com/watch?v=hD5M4a1UVz8>).
- [47] Robert Kallman, Hideaki Kimura, Jonathan Natkins i in., *H-Store: A HighPerformance, Distributed Main Memory Transaction Processing System*, „Proceedings of the VLDB Endowment”, rocznik 1, nr 2, s. 1496 – 1499, sierpień 2008 (<http://www.vldb.org/pvldb/1/1454211.pdf>).
- [48] Rich Hickey, *The Architecture of Datomic*, [infoq.com](http://infoq.com), 2 listopada 2012 (<https://www.infoq.com/articles/Architecture-Datomic>).
- [49] John Hugg, *Debunking Myths About the VoltDB In-Memory Database*, [voltdb.com](http://voltdb.com), 12 maja 2014.
- [50] Joseph M. Hellerstein, Michael Stonebraker i James Hamilton, *Architecture of a Database System*, „Foundations and Trends in Databases”, rocznik 1, nr 2, s. 141 – 259, listopad 2007 (<http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf>; <http://www.nowpublishers.com/article/Details/DBS-002>).



- [51] Michael J. Cahill, *Serializable Isolation for Snapshot Databases*, praca doktorska, University of Sydney, lipiec 2009 (<http://cahill.net.au/wp-content/uploads/2010/02/cahill-thesis.pdf>).
- [52] D. Z. Badal, *Correctness of Concurrency Control and Implications in Distributed Databases*, w: „3rd International IEEE Computer Software and Applications Conference” (COMPSAC), listopad 1979 (<http://ieeexplore.ieee.org/abstract/document/762563/>).
- [53] Rakesh Agrawal, Michael J. Carey i Miron Livny, *Concurrency Control Performance Modeling: Alternatives and Implications*, „ACM Transactions on Database Systems” (TODS), rocznik 12, nr 4, s. 609 – 654, grudzień 1987  
(<https://people.eecs.berkeley.edu/~brewer/cs262/ConcControl.pdf>;  
<https://dl.acm.org/citation.cfm?doid=32204.32220>).
- [54] Dave Rosenthal, *Databases at 14.4MHz*, [blog.foundationdb.com](http://blog.foundationdb.com), 10 grudnia 2014  
(<http://web.archive.org/web/20150427041746/http://blog.foundationdb.com/databases-at-14.4mhz>).



# Problemy z systemami rozproszonymi

*Hey I just met you  
The network's laggy  
But here's my data  
So store it maybe<sup>1</sup>.*

— Kyle Kingsbury, *Carly Rae Jepsen and the Perils of Network Partitions* (2013)

W ostatnich kilku rozdziałach powtarzającym się zagadnieniem było to, jak systemy radzą sobie z problemami. Omówiono np. przełączanie awaryjne replik („Radzenie sobie z przestojami węzłów”), opóźnienie replikacji („Problemy z opóźnieniem replikacji”) i zarządzanie współbieżnością w transakcjach („Niskie poziomy izolacji”). Lepsze zrozumienie różnych przypadków brzegowych z rzeczywistych systemów pomaga lepiej sobie z nimi radzić.

Jednak choć dużo miejsca poświęcono błędom, kilka ostatnich rozdziałów i tak miało zbyt optymistyczny wydźwięk. Rzeczywistość jest jeszcze gorsza. Teraz przyjmijmy maksymalnie pesymistyczne nastawienie i założymy, że wszystko, co *może* się nie udać, *rzeczywiście* się nie powiedzie<sup>2</sup>. Doświadczeni operatorzy systemów przyznają, że jest to sensowne założenie. Jeśli uprzejmie ich o to poprosisz, możliwe, że opowiedzą Ci jakieś przerażające historie, jednocześnie pielęgnując rany z dawnych bojów.

Praca z systemami rozproszonymi zasadniczo różni się od pisania oprogramowania przeznaczonego na jeden komputer. Podstawowa różnica polega na tym, że w środowisku rozproszonym występuje wiele nowych i interesujących możliwości wystąpienia problemów [1, 2]. W tym rozdziale poznasz próbki występujących w praktyce problemów i zrozumiesz, na czym możesz, a na czym nie możesz polegać.

Ostatecznie zadanie inżynierów polega na budowaniu systemów, które wykonują swoje zadanie (czyli spełnia gwarancje zgodnie z oczekiwaniami użytkowników) nawet wtedy, gdy wszystko zawodzi. W rozdziale 9. poznasz przykładowe algorytmy, które zapewniają takie gwarancje w systemach rozproszonych. Jednak najpierw, w tym rozdziale, musisz zrozumieć, przed jakimi wyzwaniami stoisz.

---

<sup>1</sup> „Nowy” tekst do piosenki *Call Me Maybe*: „właśnie się poznaliśmy, w sieci występują opóźnienia, ale oto są moje dane, więc może je zapisz” — *przyp. tłum.*

<sup>2</sup> Z jednym wyjątkiem — przyjmujemy, że błędy *nie są bizantyjskie* (zob. punkt „Wady bizantyjskie”).

Ten rozdział to bardzo pesymistyczny i przygnębiający przegląd tego, co może się nie powieść w systemie rozproszonym. Przyjrzyj się problemom z sieciami (punkt „Zawodne sieci”) oraz z zegarami i synchronizacją (punkt „Zawodne zegary”). Zobaczysz też, w jakim stopniu można uniknąć takich trudności. Skutki wszystkich tych problemów dezorientują, dlatego wyjaśniono też, jak myśleć o stanie systemów rozproszonych i jak analizować to, co się zdarzyło (punkt „Wiedza, prawda i kłamstwa”).

## Błędy i awarie częściowe

Gdy piszesz program przeznaczony na jeden komputer, zwykle funkcjonuje on w dość przewidywalny sposób — albo działa, albo nie działa. Oprogramowanie z błędami może wywoływać wrażenie, że komputer czasem „ma zły dzień” (ten problem często można rozwiązać, restartując maszynę), jednak kłopoty najczęściej są powodowane przez źle napisaną aplikację.

Nie istnieje fundamentalny powód, dla którego oprogramowanie na pojedynczym komputerze miałyby działać nieprawidłowo. Gdy sprzęt funkcjonuje poprawnie, te same operacje zawsze dają te same wyniki (są *deterministyczne*). Jeśli występują problemy sprzętowe (np. uszkodzenie pamięci lub obłuzowane złącze), efektem jest zwykle całkowita awaria systemu (np. komunikat *kernel panic*, „niebieski ekran śmierci”, brak możliwości uruchomienia sprzętu). Pojedynczy komputer z poprawnym oprogramowaniem zwykle albo działa w pełni poprawnie, albo w ogóle nie funkcjonuje; stan pośredni zdarza się rzadko.

Jest to celowa cecha projektu komputerów. Jeśli wystąpi wewnętrzny błąd, całkowita awaria jest preferowana od zwracania błędnego wyniku, ponieważ takie wyniki dezorientują i trudno jest sobie z nimi radzić. Dlatego komputery ukrywają rozmytą fizyczną rzeczywistość, na podstawie której są zaimplementowane, i przedstawiają wyidealizowany model systemu działającego z matematyczną precyzją. Instrukcja procesora zawsze robi to samo. Jeśli zapisujesz dane w pamięci lub na dysku, pozostają one nietknięte i nie są losowo uszkodzane. Ten cel projektowy w postaci zawsze poprawnych obliczeń pochodzi już z czasów pierwszego komputera cyfrowego [3].

Gdy piszesz oprogramowanie działające na kilku komputerach połączonych siecią, sytuacja zasadniczo się zmienia. W systemach rozproszonych oprogramowanie nie działa zgodnie z wyidealizowanym modelem. W tym kontekście nie ma wyboru i trzeba się skonfrontować ze skomplikowaną rzeczywistością świata fizycznego. A w świecie fizycznym mogą wystąpić problemy z zaskakująco wieloma rzeczami, co ilustruje następująca anegdota [4]:

W krótkim okresie czasie, w jakim zajmowałem się partycjami sieciowymi o długim czasie użytkowania w jednym centrum danych, zetknąłem się z awariami modułów dystrybucji zasilania, awariami przełączników, przypadkowym wyłączaniem i włączaniem całych szaf serwerowych, awariami całej sieci szkieletowej centrum danych, awariami zasilania w całym centrum danych i rozbiciem przez kierowcę z hipoglikemią pick-upa marki Ford o system HVAC (ogrzewania, wentylacji i klimatyzacji) centrum danych. A nawet nie jestem pracownikiem operacyjnym.

— Coda Hale

W systemie rozproszonym mogą występować uszkodzone w nieprzewidywalny sposób fragmenty, podczas gdy inne jego części działają poprawnie. Jest to tzw. *awaria częściowa*. Trudność polega na tym, że awarie częściowe są *niedeterministyczne*. Jeśli spróbujesz zrobić coś, co dotyczy wielu węzłów i sieci, czasem operacja może się powieść, a czasem w nieprzewidywalny sposób zawodzi. Zobaczysz, że możesz nawet nie *wiedzieć*, czy coś się powiodło. Wynika to z tego, że czas przesyłu komunikatu w sieci też jest niedeterministyczny!

Niedeterminizm i możliwość wystąpienia awarii częściowych sprawiają, że z systemami rozproszonymi trudno jest pracować [5].

## Przetwarzanie w chmurze i superkomputery

Istnieje spektrum sposobów budowania systemów obliczeniowych działających na dużą skalę:

- Po jednej stronie spektrum znajdują się *systemy obliczeniowe o wysokiej wydajności* (ang. *high-performance computing* — **HPC**). W tym obszarze superkomputery o tysiącach procesorów są zwykle używane do wykonywania wymagających obliczeniowo zadań z obszaru nauki, np. do prognozowania pogody lub w dynamice molekularnej (polegającej na symulowaniu ruchu atomów i cząsteczek).
- Po drugiej stronie spektrum znajduje się *przetwarzanie w chmurze*, które nie jest ściśle zdefiniowane [6], jednak często wiąże się z centrami danych obsługującymi wielu użytkowników, standardowymi komputerami połączonymi siecią IP (często Ethernetem), elastycznym przydziałem zasobów na żądanie i opłatami zależnymi od wykorzystania zasobów.
- Tradycyjne korporacyjne centra danych znajdują się gdzieś pomiędzy tymi skrajnościami.

Z tymi podejściami powiązane są bardzo odmienne sposoby obsługi błędów. W superkomputerze zadanie zwykle rejestruje od czasu do czasu punkt kontrolny (stan obliczeń) w pamięci trwałej. Jeśli jeden z węzłów zawiedzie, standardowe rozwiązanie polega na zatrzymaniu całego obciążenia roboczego w klastrze. Po naprawie uszkodzonego węzła obliczenia są wznowiane od ostatniego punktu kontrolnego [7, 8]. Dlatego superkomputer bardziej przypomina pojedynczy węzeł niż system rozproszony. Superkomputer pozwala przejść częściowej awarii w całkowitą. Jeśli któraś część zawiedzie, system pozwala na ogólną awarię (tak jak przy błędzie „kernel panic” w pojedynczej maszynie).

Ta książka dotyczy głównie systemów obsługujących usługi internetowe. Takie systemy znacznie różnią się od superkomputerów:

- Wiele aplikacji z obszaru internetu działa w trybie *online* (w tym sensie, że zawsze muszą szybko obsługiwać użytkowników). Brak dostępności takiej usługi (np. zatrzymanie klastra na czas naprawy) jest niedopuszczalny. Z kolei zadania wsadowe (w trybie *offline*), np. symulacje pogody, można stosunkowo bezproblemowo zatrzymywać i wznowiać pracę.
- Superkomputery zwykle są budowane z użyciem wyspecjalizowanego sprzętu. Każdy węzeł ma wysoką niezawodność, a węzły komunikują się z użyciem pamięci współużytkowanej i technologii RDMA (ang. *remote direct memory access*). Z kolei węzły w chmurze są budowane z użyciem standardowych maszyn, które pozwalają uzyskać analogiczną wydajność niższym kosztem (dzięki skali), ale częściej się psują.

- Sieci w dużych centrach danych są zwykle oparte na protokole IP i Ethernetie oraz działają w topologii Cłosa [9]. Superkomputery często wykorzystują wyspecjalizowane topologie sieciowe, np. wielowymiarowe kraty i torusy [10], które zapewniają wyższą wydajność dla obciążenia roboczego systemu HPC o znanych wzorcach komunikacji.
- Im system jest większy, tym większe prawdopodobieństwo uszkodzenia jednego z jego komponentów. Z czasem uszkodzone elementy są naprawiane, a psują się inne. Jednak w systemie z tysiącami węzłów można z dużym prawdopodobieństwem przyjąć, że *coś* zawsze jest uszkodzone [7]. Gdy strategia obsługi błędów polega na zwykłym poddaniu się, duży system może przez dużą część czasu przywracać stan po awariach, zamiast wykonywać użyteczną pracę [8].
- Bardzo przydatne ze względu na eksploatację i konserwację jest to, że system potrafi tolerować uszkodzenia węzłów i jako całość kontynuować pracę. Możesz wtedy np. przeprowadzać stopniową aktualizację (zob. rozdział 4.), restartując węzły jeden po drugim, podczas gdy usługa bez zakłóceń stale obsługuje użytkowników. Jeśli w chmurze jedna maszyna wirtualna źle działa, możesz ją zamknąć i zażądać nowej (z nadzieją, że nowa okaże się szybsza).
- W geograficznie rozproszonych instalacjach (dane są w nich przechowywane geograficznie blisko użytkowników, aby ograniczyć opóźnienie dostępu) komunikacja zwykle odbywa się z użyciem internetu, który — w porównaniu z sieciami lokalnymi — jest powolny i zawodny. W superkomputerach zwykle przyjmuje się, że wszystkie węzły znajdują się blisko siebie.

Jeśli chcesz używać systemów rozproszonych, musisz zaakceptować możliwość częściowych awarii i wbudować w oprogramowanie mechanizmy zapewniania odporności na błędy. Oznacza to, że musisz zbudować niezawodny system z użyciem zawodnych komponentów. W punkcie „Nie-zawodność” opisano, że nie istnieje coś takiego jak całkowita niezawodność. Dlatego trzeba zrozumieć ograniczenia tego, co można w praktyce obiecać.

Nawet w mniejszych systemach obejmujących niewiele węzłów ważne jest, aby uwzględnić częściowe awarie. W małym systemie zdarza się, że większość komponentów przez większość czasu działa prawidłowo. Jednak wcześniej czy później w którejś części systemu *wystąpi* błąd, a oprogramowanie będzie musiało jakoś sobie z nim poradzić. Obsługę błędów trzeba uwzględnić w projekcie oprogramowania, a operator aplikacji musi wiedzieć, jakiego zachowania powinien oczekiwać w reakcji na błąd.

Nierozsądnie jest zakładać, że błędy są rzadkie, i mieć nadzieję na to, iż nie wystąpią. Ważne jest, aby uwzględnić szerokie spektrum możliwych usterek (nawet stosunkowo mało prawdopodobnych) i sztucznie generować takie scenariusze w środowisku testowym, aby zobaczyć, co się stanie. W systemach rozproszonych podejrzliwość, pesymizm i paranoja są opłacalne.

## Zawodne sieci

We wprowadzeniu do części II opisano, że systemy rozproszone omawiane w tej książce to *systemy bez zasobów współużytkowanych*, czyli grupy maszyn połączonych siecią. Sieć jest jedynym sposobem komunikacji między tymi maszynami. Zakładamy, że każda maszyna ma własną pamięć i własny dysk, a dany komputer nie ma dostępu do pamięci i dysków innych urządzeń (może jedynie zgłaszać żądania do usług przez sieć).

## Budowanie niezawodnych systemów z zawodnych komponentów

Możesz się zastanawiać, czy to ma jakiś sens. Intuicyjnie może się wydawać, że system może być niezawodny tylko w takim stopniu, jak jego najbardziej zawodny komponent (*najsłabsze ogniwo*). To nieprawda. W rzeczywistości w informatyce tworzenie mniej zawodnych systemów z bardziej zawodnych podstawowych elementów to dawny pomysł [11]. Oto przykłady:

- Kody korekcji błędów umożliwiają poprawne przesyłanie danych cyfrowych kanałem komunikacyjnym, który czasem zmienia niektóre bity (np. z powodu zakłóceń radiowych w sieci bezprzewodowej) [12].
- Protokół **IP** (ang. *Internet Protocol*) jest zawodny. Może gubić pakiety, przysyłać je z opóźnieniem, duplikować, przestawiać. Protokół **TCP** (ang. *Transmission Control Protocol*) zapewnia bardziej niezawodną warstwę transportową na bazie protokołu IP. Gwarantuje, że brakujące pakiety zostaną ponownie przesłane, duplikaty usunięte, a pakiety złączone zgodnie z kolejnością ich wysłania.

Choć system może być bardziej niezawodny niż jego części, występują tu pewne ograniczenia. Na przykład kody korekcji błędów pozwalają radzić sobie z niewielką liczbą błędów na poziomie pojedynczych bitów, jeśli jednak zakłócenia sygnału są znaczne, ilość danych, które można przesłać kanałem komunikacyjnym, jest ograniczona [13]. Protokół TCP może ukrywać przed użytkownikiem utratę pakietów, duplikaty i zmianę kolejności, jednak nie wyeliminuje w magiczny sposób opóźnień w sieci.

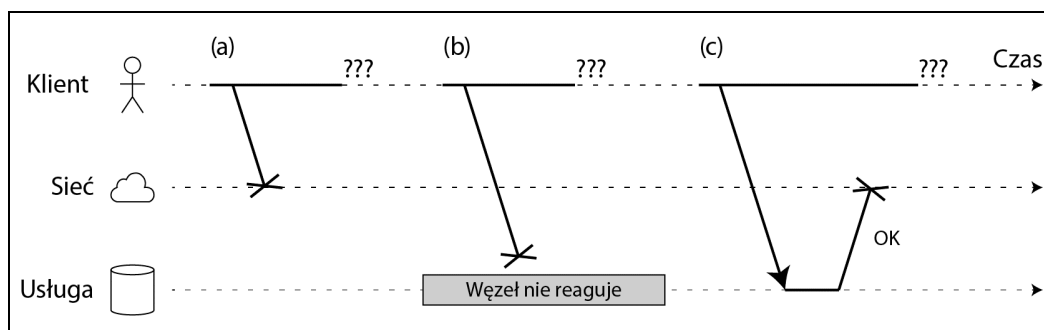
Choć mniej zawodne systemy wyższego poziomu nie są idealne, okazują się przydatne, ponieważ eliminują niektóre skomplikowane usterki z niższego poziomu. Dzięki temu zwykle łatwiej jest analizować pozostałe błędy i radzić sobie z nimi. Więcej na ten temat dowiesz się z punktu „Argument znajomości punktów końcowych”.

Model bez zasobów współużytkowanych nie jest jedynym sposobem budowania systemów, stał się jednak dominującą metodą tworzenia usług internetowych. Dzieje się tak z kilku powodów: to rozwiązanie jest stosunkowo tanie, ponieważ nie wymaga specjalnego sprzętu, można w nim wykorzystać powszechnie dostępne usługi przetwarzania w chmurze, pozwala uzyskać wysoką niezawodność dzięki nadmiarowości w wielu geograficznie rozproszonych centrach danych.

Internet i większość wewnętrznych sieci w centrach danych (często ethernetowych) to *sieci z asynchronicznym przesyłaniem pakietów*. W takich sieciach jeden węzeł przesyła komunikat (pakiet) do innego węzła, jednak sieć nie gwarantuje, kiedy te dane trafią do celu ani czy w ogóle do niego dotrą. Jeśli przesyłasz żądanie i oczekujesz na odpowiedź, może wystąpić wiele problemów (niektóre z nich są przedstawione na rysunku 8.1):

1. Żądanie mogło zostać utracone (możliwe, że ktoś odłączył kabel sieciowy).
2. Żądanie może oczekiwać w kolejce i zostać dostarczone później (możliwe, że nastąpiło przeciążenie sieci lub odbiorcy).
3. Zdalny węzeł mógł zawieść (możliwe, że uległ awarii lub został wyłączony).
4. Zdalny węzeł mógł tymczasowo przestać reagować (możliwe, że nastąpiła przerwa spowodowana długim odzyskiwaniem pamięci; zob. punkt „Przerwy w procesach”), jednak później ponownie zacznie odpowiadać.

5. Zdalny węzeł mógł przetworzyć żądanie, jednak odpowiedź została utracona w sieci (możliwe, że przełącznik sieciowy jest źle skonfigurowany).
6. Zdalny węzeł mógł przetworzyć żądanie, jednak wystąpiło opóźnienie i odpowiedź zostanie dostarczona później (możliwe, że nastąpiło przeciążenie sieci lub Twojej maszyny).



Rysunek 8.1. Jeśli przesłałeś żądanie i nie otrzymujesz odpowiedzi, nie da się stwierdzić, czy:

(a) żądanie zostało utracone, (b) zdalny węzeł nie działa lub (c) odpowiedź została utracona

Nadawca nie może nawet stwierdzić, czy pakiet został dostarczony. Jedyną możliwością dla odbiorcy jest przesłanie komunikatu z odpowiedzią, który może zostać utracony lub opóźniony. W asynchronicznej sieci te problemy są nie do odróżnienia. Jedyną dostępną informacją jest to, że nie otrzymałeś jeszcze odpowiedzi. Jeśli prześlesz żądanie do innego węzła i nie uzyskasz odpowiedzi, określenie powodu takiej sytuacji będzie *niemożliwe*.

Standardowy sposób radzenia sobie z tym problemem to stosowanie *limitu czasu*. Po pewnym czasie należy zrezygnować z oczekiwania i przyjąć, że odpowiedź nie nadejdzie. Jednak po przekroczeniu limitu czasu nadal nie wiadomo, czy zdalny węzeł otrzymał żądanie, czy nie (a jeśli żądanie znajduje się w jakiejś kolejce, może i tak zostać dostarczone do odbiorcy, nawet jeśli nadawca zrezygnował z oczekiwania).

## Błędy sieci w praktyce

Sieci komputerowe są budowane od dziesięcioleci. Można by myśleć, że do tej pory nauczyliśmy się, jak zapewniać ich niezawodność. Wydaje się jednak, że na razie jeszcze się to nie udało.

Istnieją systematyczne badania i wiele anegdotycznych dowodów na to, że problemy z siecią są zaskakująco częste — nawet w kontrolowanych środowiskach takich jak centrum danych zarządzane przez jedną firmę [14]. W jednych badaniach w średniej wielkości centrum danych wykryto ok. 12 błędów sieci na miesiąc. Połowa z nich powodowała odłączenie jednej maszyny, a połowa — odłączenie całej szafy serwerowej [15]. W innym badaniu zmierzono częstotliwość awarii komponentów takich jak przełączniki ToR (ang. *top-of-rack*), przełączniki agregacyjne i równoważniki obciążenia [16]. Odkryto, że dodanie nadmiarowego sprzętu sieciowego nie zmniejsza liczby błędów w oczekiwanym stopniu, ponieważ nie chroni to przed błędami ludzkimi (np. błędnie skonfigurowanymi przełącznikami), które są głównym powodem przestojów.



Publiczne chmury, takie jak EC2, są znane z częstych tymczasowych usterek sieci [14]. Sieci w dobrze zarządzanych prywatnych centrach danych mogą być stabilniejsze. Mimo to nikt nie jest odporny na problemy z siecią. Na przykład problem w trakcie aktualizowania oprogramowania przełącznika może spowodować zmianę konfiguracji topologii sieci. W tym czasie opóźnienie w transferze pakietów sieciowych może wynosić ponad minutę [17]. Rekiny mogą przegryźć podwodne kable i je uszkodzić [18]. Inne zaskakujące błędy to np. interfejs sieciowy, który czasem gubi wszystkie przychodzące pakiety, ale z powodzeniem wysyła pakiety wychodzące [19]. To, że połączenie sieciowe działa w jednym kierunku, nie gwarantuje jego pracy w przeciwną stronę.



### Podział sieci na partycje

Sytuację, gdy jedna część sieci jest z powodu błędu odcięta od reszty, nazywa się czasem *podziałem sieci* lub *netsplitem*. W tej książce zwykle używane jest ogólniejsze określenie *błąd sieci*, aby uniknąć mylenia takich sytuacji z opisanymi w rozdziale 6. partycjami w systemach składowania danych.

Nawet jeśli błędy sieci są w Twoim środowisku rzadkie, fakt, że *mogą* one występować, oznacza, że oprogramowanie musi umieć sobie z nimi radzić. Gdy komunikacja odbywa się z użyciem sieci, zawsze może się zdarzyć niepowodzenie. Nie da się tego uniknąć.

Jeśli obsługa błędów sieci nie jest zdefiniowana i przetestowana, mogą wystąpić różne problemy. W klastrze może np. nastąpić zakleszczenie, przez co trwale utraci on zdolność obsługi żądań — nawet po wznowieniu pracy sieci [20]. System może nawet usunąć wszystkie dane [21]. Jeśli oprogramowanie znajdzie się w nieoczekiwanej sytuacji, może zacząć działać w zupełnie niespodziewany sposób.

Obsługa błędów sieci nie musi oznaczać *odporności* na nie. Jeśli sieć zwykle działa stosunkowo niezawodnie, właściwym podejściem może być proste wyświetlanie użytkownikom komunikatu o błędzie w czasie, gdy w sieci występują problemy. Musisz jednak wiedzieć, jak oprogramowanie reaguje na problemy z siecią, i zagwarantować, że system potrafi sobie z nimi poradzić. Sensowne może być celowe wywoływanie problemów z siecią i sprawdzanie reakcji systemu (na tej zasadzie działa Chaos Monkey; zob. punkt „Niezawodność”).

## Wykrywanie błędów

Wiele systemów musi automatycznie wykrywać sprawiające problemy węzły. Oto przykłady:

- Równoważnik obciążenia musi przestać przysyłać zapytania do niesprawnego węzła (czyli *wykluczyć go z rotacji*).
- W rozproszonej bazie z replikacją z jednym liderem po awarii lidera trzeba mianować na „wyższe stanowisko” jednego z obserwatorów (zob. punkt „Radzenie sobie z przestojami węzłów”).

Niestety, niepewność dotycząca sieci sprawia, że trudno jest stwierdzić, czy dany węzeł działa. W niektórych sytuacjach możliwe jest uzyskanie informacji bezpośrednio określających, czy komponent jest sprawny:

- Jeśli możesz dotrzeć do maszyny, na której węzeł powinien działać, ale żaden proces nie oczekuje na dane w docelowym porcie (np. z powodu awarii procesu), system operacyjny w pomocny sposób zamknie lub będzie odrzucał połączenia TCP, przysyłając w odpowiedzi pakiety RST

lub FIN. Jeżeli jednak węzeł uległ awarii w trakcie przetwarzania żądania, nie da się określić, jaka część danych została przetworzona w zdalnym węźle [22].

- Jeśli proces węzła uległ awarii (lub został zamknięty przez administratora), ale system operacyjny węzła wciąż działa, skrypt może powiadomić inne węzły o awarii. Dzięki temu inny węzeł może szybko przejąć obowiązki bez konieczności oczekiwania na przekroczenie limitu czasu. Tak działa np. baza HBase [23].
- Jeśli masz dostęp do interfejsu pozwalającego zarządzać przełącznikami sieciowymi w centrum danych, możesz uruchomić zapytanie w celu wykrycia awarii połączeń na poziomie sprzętowym (stwierdzisz np., że zdalna maszyna jest wyłączona). Nie jest to możliwe, jeśli korzystasz z internetu lub współużytkowanego centrum danych bez dostępu do przełączników albo z powodu problemów z siecią nie możesz się posłużyć interfejsem administratora.
- Jeśli router stwierdzi z pewnością, że adres IP, z którym próbujesz się połączyć, jest niedostępny, może odesłać pakiet ICMP Destination Unreachable (czyli „docelowa lokalizacja jest niedostępna”). Jednak router też nie ma magicznej umiejętności wykrywania awarii. Podlega tym samym ograniczeniom co inne komponenty sieci.

Szybkie informacje zwrotne o niedostępności zdalnego węzła są przydatne, jednak nie można na nie liczyć. Nawet gdy protokół TCP potwierdzi dostarczenie pakietu, aplikacja mogła ulec awarii przed przetworzeniem go. Jeśli chcesz mieć pewność, że żądanie zostało z powodzeniem przetworzone, potrzebujesz pozytywnej odpowiedzi od samej aplikacji [24].

Jeśli natomiast coś się nie powiodło, możesz otrzymać komunikat o błędzie z któregoś poziomu stosu. Jednak zwykle trzeba przyjąć, że w ogóle nie uzyskasz odpowiedzi. Możesz kilkakrotnie ponowić próby (protokół TCP automatycznie ponawia próby, jednak możesz też zrobić to na poziomie aplikacji), odczekać na przekroczenie limitu czasu i ostatecznie uznać, że węzeł nie działa, jeśli nie odpowiedział przed upływem tego limitu.

## Limity czasu i nieograniczone opóźnienia

Skoro limit czasu jest jedynym pewnym sposobem wykrywania błędów, jak długi powinien być? Niestety, nie istnieje prosta odpowiedź na to pytanie.

Długi limit czasu oznacza długie oczekiwanie do momentu uznania węzła za niesprawny (a w tym czasie użytkownicy mogą być zmuszeni do oczekiwania lub oglądania komunikatu o błędzie). Krótki limit czasu pozwala szybciej wykryć błędy, jednak zwiększa ryzyko niesłusznego uznania węzła za niesprawny, gdy tylko tymczasowo jego działanie zaczęło działać wolniej (np. z powodu skokowego wzrostu obciążenia w węźle lub sieci).

Przedwczesne uznanie węzła za niesprawny to problem. Jeśli węzeł w rzeczywistości działa i właśnie wykonuje jakąś pracę (np. wysyła e-mail), a inny węzeł przejmie jego zadania, operacje mogą zostać wykonane dwukrotnie. To zagadnienie jest szczegółowo opisane w punkcie „Wiedza, prawda i kłamstwa” oraz w rozdziałach 9. i 11.

Gdy węzeł zostaje uznany za niesprawny, jego zadania trzeba przekazać innym węzłom, co dodatkowo zwiększa obciążenie tych jednostek i sieci. Jeśli system już ma problemy z wysokim obciążeniem, przedwczesne stwierdzenie niesprawności węzła może nasilić trudności. Możliwe, że węzeł w rzeczywistości działa i tylko reaguje powoli z powodu przeciążenia. Przekazanie obciążenia tego

węzła innym jednostkom może spowodować awarię kaskadową (w skrajnym przypadku wszystkie węzły uznają pozostałe za niesprawne i cały system przestanie działać).

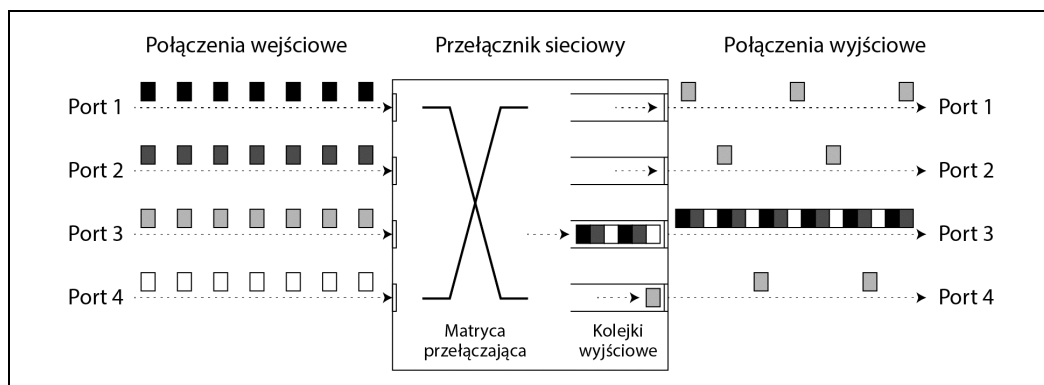
Wyobraź sobie fikcyjny system z siecią gwarantującą nieprzekraczanie maksymalnego opóźnienia przesyłu pakietów. Każdy pakiet albo jest dostarczany w czasie  $d$ , albo zostaje utracony, jednak czas przesyłu nigdy nie przekracza  $d$ . Ponadto przyjmij, że możesz zapewnić, iż nieuszkodzony węzeł zawsze obsługuje żądanie w czasie nie dłuższym niż  $r$ . Wtedy możesz zagwarantować, że odpowiedź na każde udanie obsłużone żądanie przychodzi w czasie nieprzekraczającym  $2d + r$ . Jeśli nie otrzymasz odpowiedzi w tym czasie, wiesz, że albo sieć, albo zdalny węzeł nie działają. Wtedy sensownym limitem czasu jest  $2d + r$ .

Niestety, w większości używanych systemów nie można zapewnić żadnej z tych gwarancji. W sieciach asynchronicznych występują *nieograniczone opóźnienia* (to oznacza, że sieć próbuje dostarczyć pakiety tak szybko, jak jest to możliwe, jednak nie istnieje górny limit czasu potrzebnego na przesłanie pakietu). Ponadto większość implementacji serwera nie może zagwarantować obsługi żądań w określonym czasie maksymalnym (zob. punkt „Gwarancje czasu zwrócenia odpowiedzi”). Na potrzeby wykrywania awarii nie wystarcza, że system przez większość czasu działa szybko. Jeśli limit czasu jest niski, wystarczy przejściowy skok w czasie przesyłania danych, aby zakłócić równowagę systemu.

## Przeciążenie sieci i kolejki

Gdy prowadzisz samochód, czas podróży siecią dróg często zależy głównie od natężenia ruchu. Podobnie zmienność w opóźnieniu pakietów w sieciach komputerowych najczęściej jest spowodowana kolejkami [25]:

- Jeśli kilka różnych węzłów jednocześnie próbuje wysłać pakiety do tej samej lokalizacji, przełącznik sieciowy musi umieszczać pakiety w kolejce i przekazywać je do docelowego połączenia sieciowego jeden po drugim (ilustruje to rysunek 8.2). Gdy połączenie sieciowe jest obciążone, możliwe, że pakiet będzie musiał przez pewien czas czekać na wolne miejsce (taka sytuacja to *przeciążenie sieci*). Jeżeli danych przychodzących jest tak dużo, że kolejka przełącznika zostaje zapełniona, pakiet zostaje odrzucony, dlatego trzeba przesłać go jeszcze raz, choć sieć działa poprawnie.
- Gdy pakiet dotrze do docelowej maszyny, to jeśli wszystkie rdzenie procesora są zajęte, system operacyjny umieszcza przychodzące żądanie z sieci w kolejce. Żądanie pozostaje w niej do momentu, gdy aplikacja będzie gotowa je obsłużyć. To, jak długo to potrwa, zależy od obciążenia maszyny.
- W środowiskach z wirtualizacją działający system operacyjny jest często wstrzymywany na dziesiątki milisekund, kiedy inna maszyna wirtualna używa rdzenia procesora. W tym czasie maszyna wirtualna nie może przetwarzać żadnych danych z sieci, dlatego przychodzące dane są umieszczane przez kontroler maszyny wirtualnej w kolejce (buforze) [26], co dodatkowo zwiększa zmienność opóźnień w sieci.
- Protokół TCP stosuje *kontrolę przepływu* (inne określenia to *unikanie przeciążenia*). Polega to na tym, że węzeł ogranicza szybkość przesyłania danych, aby uniknąć przeciążenia połączenia sieciowego lub odbiorcy danych [27]. Oznacza to dodatkowe kolejki u nadawcy jeszcze przed umieszczeniem danych w sieci.



Rysunek 8.2. Jeśli kilka maszyn przesyła dane sieciowe do tej samej docelowej jednostki, kolejka przełącznika może się zapęłnić. Tu porty 1, 2 i 4 próbują przesyłać pakiety do portu 3

Ponadto protokół TCP uznaje pakiet za utracony, jeśli nie otrzyma potwierdzenia w określonym limicie czasu (wyznaczanym na podstawie zaobserwowanego czasu przesyłu komunikatów), a utracone pakiety są automatycznie ponownie przesyłane. Choć aplikacja nie widzi utraty i ponownego przesyłania pakietów, odczuwa wynikowe opóźnienie (oczekiwanie na upływ limitu czasu, a następnie na potwierdzenie otrzymania ponownie przesłanego pakietu).

Wszystkie wymienione czynniki wpływają na zmienność opóźnienia w sieci. Opóźnienia związane z kolejkami bywają bardzo zmienne zwłaszcza wtedy, gdy system pracuje blisko maksymalnych możliwości. System z dużą ilością wolnych zasobów może sprawić pobierać elementy z kolejki, natomiast w mocno obciążonym systemie bardzo szybko mogą powstawać długie kolejki.

W publicznych chmurach i centrach danych z wieloma użytkownikami zasoby są współużytkowane przez wielu klientów. Połączenia sieciowe i przełączniki, a nawet karty sieciowe i procesory poszczególnych maszyn (gdy używane są maszyny wirtualne) są współużytkowane. Wsadowe obciążenie robocze, np. w algorytmie MapReduce (zob. rozdział 10.), może łatwo przeciążyć połączenia sieciowe. Ponieważ nie możesz kontrolować (ani zobaczyć) tego, jak inni klienci wykorzystują współużytkowane zasoby, opóźnienia sieciowe mogą być wysoce zmienne, jeśli ktoś w Twoim pobliżu zużywa wiele zasobów (*hałaśliwy sąsiad*) [28, 29].

## Protokoły TCP i UDP

Niektóre wrażliwe na opóźnienia aplikacje, np. do obsługi wideokonferencji lub telefonii **VoIP** (ang. *Voice over IP*), wykorzystują protokół UDP zamiast TCP. Jest to kompromis między niezawodnością a zmiennością opóźnień. Ponieważ protokół UDP nie obejmuje kontroli przepływu i nie retransmituje utraconych pakietów, unika niektórych przyczyn opóźnień w sieci (choć i tak jest podatny na kolejki w przełączniku i opóźnienia związane z szeregowaniem).

UDP jest dobrym wyborem w sytuacjach, gdy opóźnione dane są bezwartościowe. Na przykład w połączeniach VoIP prawdopodobnie nie ma czasu na ponowne przesyłanie utraconego pakietu, aby zdążyć wygenerować głos w głośniku. Wtedy ponowne przesyłanie pakietu nie ma sensu. Aplikacja musi zamiast tego zapęłnić związane z utraconym pakietem okno czasowe ciszą (co powoduje krótką przerwę) i przejść dalej w strumieniu. Ponowienie próby następuje wtedy w „warstwie ludzkiej” („Czy mógłbyś to powtórzyć? Przez chwilę nie było nic słychać”).

W takich środowiskach limit czasu można ustalić wyłącznie eksperymentalnie — mierząc rozkład czasów przesyłu komunikatów w sieci przez dłuższy okres i na wielu maszynach, aby określić oczekiwaną zmienność opóźnień. Następnie na podstawie cech aplikacji możesz ustalić odpowiedni kompromis między opóźnieniem w wykrywaniu awarii a ryzykiem przedwczesnego przekroczenia limitu czasu.

Jeszcze lepszy od używania skonfigurowanego stałego limitu jest ciągły pomiar przez systemy czasów odpowiedzi i ich zmienności (ang. *jitter*) oraz automatyczne dostosowywanie limitu czasu na podstawie zaobserwowanego rozkładu czasów odpowiedzi. Można to zrobić za pomocą narzędzia Phi Accrual Failure Detector [30], używanego np. w bazach Akka i Cassandra [31]. Podobnie działają limity czasu używane na potrzeby retransmisji w protokole TCP [27].

## Sieci synchroniczne i asynchroniczne

Systemy rozproszone byłyby znacznie prostsze, gdyby mogły polegać na tym, że sieć dostarczy pakiety z nieprzekraczalnym maksymalnym opóźnieniem i nie zgubi pakietów. Dlaczego nie można rozwiązać tego problemu na poziomie sprzętowym i zapewnić niezawodności sieci w taki sposób, aby nie trzeba było przejmować się tym problemem na poziomie oprogramowania?

Aby odpowiedzieć na to pytanie, warto porównać sieci w centrum danych z tradycyjnymi stacjonarnymi sieciami telefonicznymi (niekomórkowymi, nie VoIP), które są wysoce niezawodne. Utrata fragmentów dźwięku i przerwane połączenia zdarzają się bardzo rzadko. Rozmowa telefoniczna wymaga utrzymywania przez cały czas niskiego opóźnienia między punktami końcowymi i wystarczającej przepustowości, by przesyłać próbki dźwięku z głosem. Czy nie byłoby wygodnie uzyskać podobną niezawodność i przewidywalność w sieciach komputerowych?

Gdy rozmawiasz przez sieć telefoniczną, tworzony jest *obwód*. Połączeniu przydzielana jest stała, gwarantowana przepustowość na całej trasie między rozmówcami. Ten obwód pozostaje aktywny do momentu zakończenia rozmowy [32]. Na przykład sieć ISDN działa ze stałą szybkością 4000 ramek na sekundę. Po nawiązaniu połączenia każdej ramce (w obu kierunkach) przydzielanych jest 16 bitów. Tak więc przez czas trwania połączenia każda strona ma gwarancję, że będzie mogła co 250 ms przesłać dokładnie 16 bitów danych dźwiękowych [33, 34].

Tak działa sieć *synchroniczna*. Gdy dane są przekazywane przez kilka routerów, nie są narażone na kolejki, ponieważ 16 bitów dla połączenia zostało już zarezerwowanych w następnej stacji w sieci. A ponieważ nie ma kolejek, maksymalne opóźnienie między punktami końcowymi jest stałe. Jest to *opóźnienie ograniczone*.

### Czy nie można zapewnić przewidywalności opóźnienia w sieci?

Zauważ, że obwód w sieci telefonicznej bardzo się różni od połączenia TCP. Obwód ma stałą ilość zarezerwowanej przepustowości, niedostępną dla innych jednostek przez czas działania obwodu. Natomiast w połączeniu TCP pakiety oportunistycznie wykorzystują dostępną przepustowość sieci. Możesz przekazywać protokołowi TCP dane w blokach o zmiennej wielkości (np. e-mail lub stronę internetową), a protokół spróbuje przesłać dane w możliwie najkrótszym czasie. Gdy połączenie TCP jest nieaktywne, nie wykorzystuje przepustowości<sup>3</sup>.

---

<sup>3</sup> Wyjątkiem są okresowo przesyłane pakiety sygnalizujące aktywność, jeśli ta funkcja protokołu TCP jest włączona.

Gdyby sieci w centrum danych i internet były sieciami z komutacją łączy, po ustanowieniu obwodu możliwe byłoby określenie gwarantowanego maksymalnego czasu przesyłu komunikatów. Jest jednak inaczej. Ethernet i IP są protokołami z komutacją pakietów narażonymi na powstawanie kolejek i tym samym nieograniczonych opóźnień w sieci. W tych protokołach obwody nie występują.

Dlaczego w sieciach centrów danych i internecie stosowana jest komutacja pakietów? Wynika to z tego, że te sieci są zoptymalizowane pod kątem *przesyłania pakietów partiami*. Obwód dobrze nadaje się do połączeń dźwiękowych lub wideo, które przez czas trwania rozmowy muszą przekazywać stosunkowo stałą liczbę bitów na sekundę. Z kolei żądanie strony internetowej, przesyłanie e-maila lub transfer pliku nie wymagają określonej przepustowości. Celem jest tylko jak najszybsze wykonanie takich operacji.

Jeśli chcesz przesłać plik za pomocą obwodu, musisz oszacować przepustowość. Zbyt niskie szacunki sprawią, że transfer będzie niepotrzebnie powolny i możliwości sieci pozostaną niewykorzystane. Jeżeli ustalisz za wysoką przepustowość, obwodu nie będzie można utworzyć (ponieważ sieć nie może pozwolić na dodanie obwodu, gdy nie może zagwarantować przydziału żądanej przepustowości). Dlatego używanie obwodów do przesyłania pakietów partiami marnuje możliwości sieci i niepotrzebnie spowalnia transfer. Z kolei protokół TCP dynamicznie dostosowuje szybkość transferu danych do dostępnych możliwości sieci.

Próbowano budować sieci hybrydowe obsługujące zarówno komutację łączy, jak i komutację pakietów, np. z użyciem standardu ATM<sup>4</sup>. Pewnymi podobieństwami cechuje się standard Infini-Band [35]. Obsługuje on zarządzanie przepływem danych między punktami końcowymi w warstwie łączy danych, co zmniejsza konieczność stosowania kolejek w sieci. Nadal jednak mogą występować opóźnienia spowodowane przeciążeniem łączy [36]. Dzięki starannemu zarządzaniu *jakością usług* (z priorytetyzowaniem i planowaniem transferu pakietów) oraz *kontrolą dostępu* (ograniczaniem nadawców) można zasymulować komutację łączy w sieciach pakietowych lub zapewnić statystycznie ograniczone opóźnienie [25, 32].

Jednak tego rodzaju jakość usług nie jest gwarantowana we współużytkowanych centrach danych i chmurach publicznych oraz w ramach komunikacji w internecie<sup>5</sup>. Obecnie stosowane technologie nie pozwalają na gwarantowanie opóźnień lub niezawodności sieci. Trzeba zakładać, że przeciążenie sieci, kolejki i nieograniczone opóźnienia będą występować. Nie istnieje więc „prawidłowy” poziom limitu czasu — trzeba go ustalać eksperymentalnie.

---

<sup>4</sup> ATM (ang. *Asynchronous Transfer Mode*) był w latach 80. konkurentem Ethernetu [32], jednak nie był szeroko stosowany poza przełącznikami głównymi w sieciach telefonicznych. Nie ma on nic wspólnego z bankomatami, po angielsku także nazywanymi ATM. Możliwe, że w jakimś świecie równoległym internet jest oparty na czymś podobnym do sieci ATM. W takim świecie internetowe połączenia wideo są zapewne dużo stabilniejsze niż w naszym, ponieważ nie są narażone na utratę i opóźnienia pakietów.

<sup>5</sup> Umowy między dostawcami usług internetowych i tworzenie tras z użyciem protokołu BGP (ang. *Border Gateway Protocol*) bardziej przypominają komutację łączy niż działanie protokołu IP. Na tym poziomie możliwy jest zakup dedykowanej przepustowości. Jednak trasowanie w internecie odbywa się na poziomie sieci, a nie pojedynczych połączeń między hostami i działa w znacznie dłuższej skali czasowej.

## Opóźnienia i wykorzystanie zasobów

Ogólnie możesz traktować zmienne opóźnienia jak skutek dynamicznego przydziału zasobów.

Załóżmy, że między dwoma przełącznikami telefonicznymi znajduje się przewód potrafiący jednocześnie obsłużyć 10 tys. rozmów. Każdy obwód wykorzystujący ten przewód zajmuje jeden ze slotów. Przewód możesz więc traktować jak zasób, który może być jednocześnie współużytkowany przez 10 tys. użytkowników. Ten zasób jest dzielony *statycznie*. Nawet jeśli jesteś jedynym rozmówcą korzystającym z przewodu i wszystkie pozostałe 9999 slotów pozostaje wolnych, Twojemu obwodowi przydzielona zostaje ta sama stała część przepustowości jak wtedy, gdy zajęty jest cały przewód.

Natomiast internet dzieli przepustowość sieci *dynamicznie*. Nadawcy rywalizują ze sobą, aby jak najszybciej przesłać swoje pakiety przewodem, a przełączniki sieciowe z momentu na moment decydują o tym, które pakiety przesłać (decydują o przydziale przepustowości). Wadą tego podejścia są kolejki, a zaletą — maksymalne wykorzystanie przewodu. Koszty utrzymania przewodu są stałe, dlatego jeśli lepiej go wykorzystasz, przesyłanie w nim bajtów będzie tańsze.

Podobnie jest z procesorami. Jeśli dynamicznie przydzielasz każdy rdzeń procesora kilku wątkom, jeden wątek musi czasem czekać w kolejce systemu operacyjnego, gdy działa inny wątek. Wątki mogą być więc wstrzymywane przez zmienny czas. Jednak to rozwiązanie pozwala lepiej wykorzystać sprzęt niż w sytuacji, gdy każdemu wątkowi statycznie przydzielisz określoną liczbę cykli procesora (zob. punkt „Gwarancje czasu przesłania odpowiedzi”). Lepsze wykorzystanie sprzętu jest też ważnym powodem do stosowania maszyn wirtualnych.

W niektórych środowiskach można uzyskać gwarancje opóźnień, jeśli przydział zasobów jest statyczny (czyli przy przydziale sprzętu i przepustowości na wyłączność). Dzieje się to jednak kosztem niższego wykorzystania możliwości sprzętu, co podnosi koszty. Z kolei współużytkowanie zasobów z ich dynamicznym przydziałem skutkuje lepszym wykorzystaniem sprzętu, jest więc tańsze, jednak wadą jest zmienne opóźnienie.

Zmienne opóźnienie w sieciach nie jest prawem natury, a prostym skutkiem kompromisu między kosztami a korzyściami.

## Zawodne zegary

Zegary i czas są ważne. Aplikacje potrzebują zegarów w różnych obszarach, aby móc odpowiadać na następujące pytania:

1. Czy limit czasu dla żądania już upłynął?
2. Ile wynosi percentyl 99% czasów odpowiedzi w danej usłudze?
3. Ile zapytań na sekundę usługa średnio obsługiwała w ostatnich pięciu minutach?
4. Jak dużo czasu użytkownik spędził w witrynie?
5. Kiedy opublikowany został dany artykuł?
6. O jakiej godzinie i jakiego dnia należy przesłać e-mail z przypomnieniem?
7. Kiedy wygasa dany element w pamięci podręcznej?
8. Jaki jest znacznik czasu dla określonego komunikatu o błędzie z pliku dziennika?

Punkty od 1. do 4. dotyczą *czasu trwania* (czyli okresu od przesłania żądania do otrzymania odpowiedzi), natomiast punkty od 5. do 8. opisują *punkty w czasie* (zdarzenia zachodzące danego dnia o określonej godzinie).

W systemach rozproszonych czas jest skomplikowanym zagadnieniem, ponieważ komunikacja nie odbywa się natychmiastowo. Czas otrzymania komunikatu jest zawsze późniejszy niż moment jego wysłania, jednak z powodu opóźnień w sieci nie wiadomo, jak długo trwa transfer. Dlatego gdy zaangażowanych jest wiele maszyn, czasem trudno jest ustalić kolejność zdarzeń.

Ponadto każda maszyna w sieci używa własnego zegara sprzętowego (jest nim zwykle oscylator kwarcowy). Precyzja tych urządzeń nie jest doskonała, dlatego każda maszyna stosuje własny czas, który może się nieco różnić od czasu z innych urządzeń. Możliwa jest synchronizacja zegarów w pewnym zakresie. Najczęściej stosowany do tego mechanizm to **NTP** (ang. *Network Time Protocol*), który umożliwia dopasowanie zegara komputera do czasu podawanego przez grupę serwerów [37]. Z kolei serwery pobierają czas z bardziej precyzyjnego źródła, np. odbiornika GPS.

## Zegary monotoniczne a zegary czasu rzeczywistego

Obecnie w komputerach występują przynajmniej dwa różne rodzaje zegarów: *zegary czasu rzeczywistego* i *zegary monotoniczne*. Choć oba mierzą czas, ważne jest, by odróżniać je od siebie, ponieważ służą odmiennym celom.

### Zegary czasu rzeczywistego

Zegar czasu rzeczywistego robi to, czego intuicyjnie oczekujemy od zegara: zwraca aktualną datę i czas zgodnie z określonym kalendarzem (jest to *czas zegarowy*). Na przykład wywołanie `clock_gettime(CLOCK_REALTIME)` w Linuksie<sup>6</sup> lub `System.currentTimeMillis()` w Javie zwracają liczbę sekund (lub milisekund) od początku *epoki Unixa*, czyli północy 1 stycznia 1970 r. czasu UTC (zgodnie z kalendarzem gregoriańskim i bez uwzględniania sekund przestępnych). W niektórych systemach punktem odniesienia jest inna data.

Zegary czasu rzeczywistego są zwykle synchronizowane za pomocą NTP, tzn. znacznik czasu z jednej maszyny (w idealnych warunkach) oznacza to samo co w innym komputerze. Takie zegary mają jednak pewne opisane w następnym punkcie osobliwości. Przede wszystkim chodzi o to, że jeśli zegar lokalny zanadto wyprzedza serwer NTP, może zostać siłowo przestawiony. Wygląda to wtedy tak, jakby cofnął się w czasie. Takie przeskoki oraz częste ignorowanie sekund przestępnych sprawiają, że zegary czasu rzeczywistego nie nadają się do pomiaru upływu czasu [38].

Zegary czasu rzeczywistego w przeszłości miały też niską precyzję. Na przykład w starszych wersjach systemu Windows przeskakiwały w przód po 10 ms [39]. W nowszych systemach ten problem jest mniej istotny.

---

<sup>6</sup> Choć w nazwie występuje człon *czasu rzeczywistego*, takie zegary nie mają nic wspólnego z systemami operacyjnymi czasu rzeczywistego. Jest to opisane w punkcie „Gwarancje czasu zwracania odpowiedzi”.



## Zegary monotoniczne

Zegar monotoniczny nadaje się do pomiaru czasu trwania (przedziałów czasu), np. limitów czasu lub czasu odpowiedzi usługi. Zgodnie z zegarem monotonicznym działają wywołania `clock_gettime` ↪ (`CLOCK_MONOTONIC`) w Linuksie i `System.nanoTime()` w Javie. Nazwa tych zegarów pochodzi od tego, że gwarantowane jest, iż zawsze poruszają się one do przodu (podczas gdy zegary czasu rzeczywistego mogą przeskakiwać wstecz).

W dowolnym momencie możesz sprawdzić wartość zegara monotonicznego, wykonać jakieś operacje i później ponownie sprawdzić zegar. *Różnica* między dwiema wartościami informuje o tym, ile czasu upłynęło między jednym sprawdzeniem a drugim. Jednak *bezwzględna* wartość zegara nie ma określonego znaczenia. Może nią być liczba nanosekund od momentu uruchomienia komputera lub coś równie arbitralnego. Przede wszystkim nie ma sensu porównywanie wartości zegara monotonicznego z dwóch różnych komputerów, ponieważ wartości te nie oznaczają tego samego.

Na serwerze z wieloma gniazdami procesorów dla każdego procesora może być używany odrębny zegar, który nie zawsze jest zsynchronizowany z pozostałymi procesorami. System operacyjny kompensuje rozbieżności i próbuje zapewniać monotoniczny obraz zegara wątkom aplikacji, nawet jeśli są one wykonywane w różnych procesorach. Warto jednak sceptycznie traktować takie gwarancje monotoniczności [40].

Serwer NTP może dostosować szybkość zegara monotonicznego (wykonując *dostrajanie*), jeśli wykryje, że lokalny oscylator komputera działa szybciej lub wolniej niż na serwerze NTP. Protokół NTP domyślnie dopuszcza przyspieszanie lub spowalnianie zegara do 0,05%, przy czym nie-dopuszczalne jest przesuwanie zegarów monotonicznych do przodu lub do tyłu. Precyzja zegarów monotonicznych jest zwykle wysoka. W większości systemów potrafią one mierzyć przedział czasu na poziomie mikrosekund lub niższym.

W systemie rozproszonym używanie zegara monotonicznego do pomiaru upływu czasu (np. limitów czasu) jest zwykle akceptowalne, ponieważ takie zegary nie wymagają synchronizacji między zegarami z różnych węzłów i nie są wrażliwe na drobne odchylenia pomiarów.

## Synchronizacja i precyzja zegarów

Zegary monotoniczne nie wymagają synchronizacji, jednak aby zegary czasu rzeczywistego były przydatne, trzeba je nastawić zgodnie z serwerem NTP lub innym zewnętrznym źródłem czasu. Niestety, metody uzyskiwania poprawnego czasu z zegara nie są tak niezawodne i precyzyjne, jak można by oczekiwać. Zegary sprzętowe i serwery NTP bywają kapryśne. Oto kilka przykładów:

- Zegar kwarcowy w komputerze nie jest precyzyjny. Występują w nim *odchylenia* (działa szybciej lub wolniej, niż powinien). Odchylenia zegara zależą od temperatury maszyny. Google przyjmuje dla swoich serwerów odchylenia na poziomie 200 ppm (jednostek na milion), co odpowiada 6 ms dla zegara zsynchronizowanego z serwerem co 30 s lub 17 s dla zegara zsynchronizowanego raz dziennie. To odchylenie ogranicza optymalną precyzję — nawet jeśli wszystko działa prawidłowo.
- Jeśli zegar komputera zanadto odbiega od czasu z serwera NTP, może nastąpić odmowa synchronizacji lub siłowe przestawienie lokalnego zegara [37]. Aplikacje obserwujące czas przed przestawieniem i po tej operacji mogą zobaczyć, jak zegar nagle przeskakuje do przodu lub do tyłu.

- Jeśli węzeł zostanie przypadkowo odcięty przez zaporę od serwerów NTP, błąd w konfiguracji może przez pewien czas pozostać niezauważony. Anegdotyczne dowody wskazują na to, że taka sytuacja zdarza się w praktyce.
- Jakość synchronizacji z serwerem NTP zależy od opóźnienia w sieci. Dlatego w przeciążonej sieci ze zmiennym opóźnieniem pakietów precyzja tego mechanizmu jest ograniczona. W jednym eksperymencie wykazano, że minimalny błąd przy synchronizacji za pomocą internetu wynosi 35 ms [42], choć okresowe skoki opóźnienia powodowały błędy na poziomie ok. sekundy. W zależności od konfiguracji opóźnienia w dużej sieci mogą spowodować, że klient serwera NTP zrezygnuje z synchronizacji.
- Niektóre serwery NTP działają błędnie lub są źle skonfigurowane i zgłaszają czas z błędem kilku godzin [43, 44]. Klienci serwerów NTP są dość odporne, ponieważ pobierają dane z paru serwerów i ignorują skrajne wartości. Mimo to niepokojące jest opieranie poprawności systemu na czasie otrzymanym od zewnętrznej jednostki z internetu.
- Sekundy przestępne powodują, że minuta może mieć 59 lub 61 s. Zakłóca to dotyczące czasu założenia w systemach, które nie są zaprojektowane z myślą o sekundach przestępnych [45]. To, że sekundy przestępne doprowadziły do awarii w wielu dużych systemach [38, 46], pokazuje, jak łatwo jest przyjąć w systemie błędne założenia dotyczące zegarów. Najlepszym sposobem obsługi sekund przestępnych może być „kłamanie” przez serwery NTP i stopniowe wprowadzanie sekund przestępnych w ciągu dnia (czyli *rozkładanie* sekund przestępnych) [47, 48]. Jednak w praktyce serwery NTP działają w różny sposób [49].
- W maszynach wirtualnych także zegar sprzętowy działa wirtualnie. Powoduje to dodatkowe wyzwania w aplikacjach wymagających precyzyjnego pomiaru czasu [50]. Gdy rdzeń procesora jest współużytkowany przez maszyny wirtualne, każda maszyna jest wstrzymywana na dziesiątki milisekund w czasie pracy innej maszyny. Z perspektywy aplikacji ta przerwa jest odbierana jak nagły przeskok zegara wprzód [26].
- Jeśli uruchamiasz oprogramowanie w urządzeniach, nad którymi nie masz pełnej kontroli (np. w urządzeniach mobilnych lub wbudowanych), prawdopodobnie w ogóle nie możesz ufać zegarowi sprzętowemu. Niektórzy użytkownicy celowo ustawiają w zegarze sprzętowym błędną datę i godzinę — np. w celu ominięcia ograniczeń czasowych w grach. W efekcie czas w zegarze może być znacznie przesunięty wstecz lub do przodu względem rzeczywistości.

Możesz uzyskać bardzo wysoką precyzję zegara, jeśli jest do dla Ciebie tak ważne, by zainwestować znaczne środki. Na przykład dyrektywa unijna MiFID II dla instytucji finansowych wymaga, by wszystkie fundusze HFT synchronizowały swoje zegary z dokładnością do 100 ms względem czasu UTC. Ma to pomóc w analizie anomalii rynkowych takich jak flash crash i w wykrywaniu manipulacji [51].

Taką precyzję można uzyskać za pomocą odbiorników GPS, protokołu **PTP** (ang. *Precision Time Protocol*) [52] oraz starannego wdrażania i monitorowania rozwiązań. Wymaga to jednak znacznego wysiłku i dużej wiedzy. Ponadto synchronizacja zegara może się z wielu powodów nie udać. Jeśli demon NTP jest błędnie skonfigurowany lub gdy zaporą blokuje połączenie z serwerem NTP, błąd zegara spowodowany odchyleniami może szybko stać się duży.

## Poleganie na zegarach synchronizowanych

Problem z zegarami polega na tym, że choć wydają się proste i łatwe w użyciu, związanych jest z nimi zaskakująco dużo pułapek. Dzień może nie liczyć dokładnie 86 400 s, zegary czasu rzeczywistego mogą „chodzić do tyłu”, a czas w jednym węźle może się znacznie różnić od czasu w innym.

Wcześniej w rozdziale opisano gubienie pakietów w sieci i arbitralnie długie opóźnienia. Choć przez większość czasu sieci działają poprawnie, oprogramowanie trzeba projektować zgodnie z założeniem, że czasami sieć będzie działać nieprawidłowo. Oprogramowanie musi sprawnie radzić sobie z takimi błędami. To samo dotyczy zegarów. Choć przez większość czasu pracują poprawnie, niezawodne oprogramowanie musi być gotowe na radzenie sobie z błędnymi zegarami.

Problem po części wynika z tego, że łatwo jest przeoczyć niepoprawną pracę zegarów. Jeśli procesor maszyny jest uszkodzony lub gdy sieć jest błędnie skonfigurowana, zapewne w ogóle nie będą działać. Dlatego problem zostanie szybko wykryty i naprawiony. Jeśli jednak zegar działa nieprawidłowo lub klient NTP jest źle skonfigurowany, większość komponentów będzie na pozór działała prawidłowo — nawet gdy wskazania zegara są coraz bardziej oddalone od rzeczywistości. Jeżeli oprogramowanie polega na precyzyjnie zsynchronizowanym zegarze, efektem częściej będzie wtedy niezgłaszana i subtelna utrata danych niż gwałtowna awaria [53, 54].

Dlatego jeśli używasz oprogramowania wymagającego zsynchronizowanych zegarów, koniecznie powinieneś starannie monitorować różnice między zegarami z wszystkich maszyn. Każdy węzeł, którego zegar zanadto odbiega od pozostałych, należy uznać za niesprawny i usunąć z klastra. Takie monitorowanie gwarantuje, że zauważysz uszkodzone zegary, zanim wyrządzą poważne szkody.

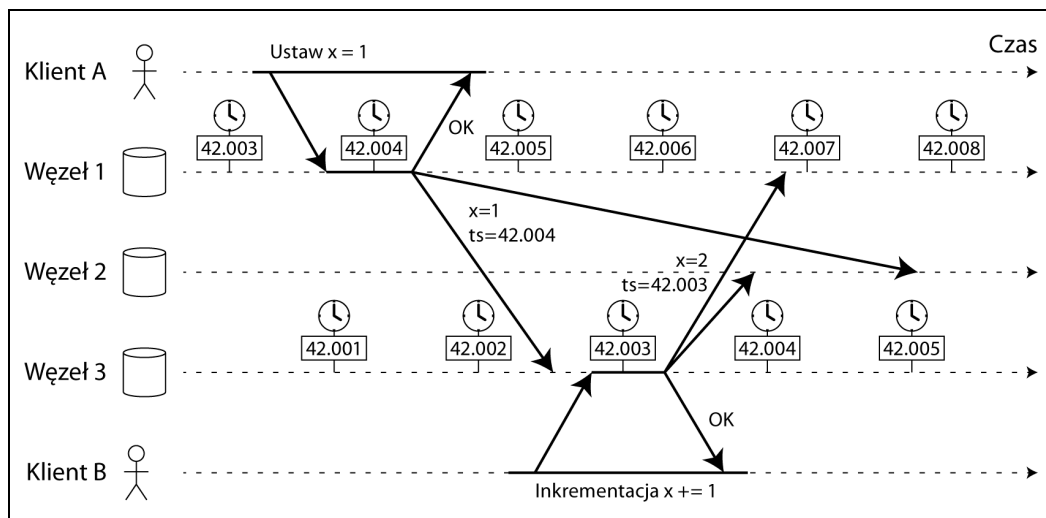
### Używanie znaczników czasu do porządkowania zdarzeń

Rozważmy pewną sytuację, w której poleganie na zegarach jest kuszące, ale niebezpieczne — porządkowanie zdarzeń w wielu węzłach. Na przykład jeśli dwa klienty zapisują dane w rozproszonej bazie, który zrobił to jako pierwszy? Który zapis jest najnowszy?

Na rysunku 8.3 pokazane jest niebezpieczne wykorzystanie zegarów czasu rzeczywistego w bazie z replikacją z wieloma liderami (ten przykład jest podobny do scenariusza z rysunku 5.9). Klient A zapisuje  $x = 1$  w węźle 1. Ten zapis jest replikowany w węźle 3. Klient B zwiększa wartość  $x$  w węźle 3 (teraz  $x = 2$ ). W ostatnim kroku oba zapisy są replikowane w węźle 2.

Na rysunku 8.3 zapis po replikacji w innych węzłach jest opatrzony znacznikiem czasu zgodnym z zegarem czasu rzeczywistego z węzła, gdzie zapis wprowadzono po raz pierwszy. W tym przykładzie synchronizacja zegara jest bardzo dobra. Różnica między węzłami 1 i 3 wynosi mniej niż 3 ms, co jest wynikiem lepszym od tego, jakiego możesz oczekiwać w praktyce.

Mimo to na rysunku 8.3 znaczniki czasu nie wyznaczają poprawnej kolejności zdarzeń. Zapis  $x = 1$  ma znacznik czasu 42,004 s, natomiast znacznik czasu zapisu  $x = 2$  to 42,003 s, choć operacja  $x = 2$  z pewnością została wykonana później. Gdy węzeł 2 otrzymuje te dwa zdarzenia, błędnie przyjmuje, że  $x = 1$  to nowsza operacja, i ignoruje zapis  $x = 2$ . W efekcie operacja zwiększania wartości wykonana przez klienta B zostaje utracona.



Rysunek 8.3. Zapis od klienta B jest przyczynowo późniejszy niż zapis od klienta A, jednak zapis od klienta B ma wcześniejszy znacznik czasu

Ta strategia rozwiązywania konfliktów to *ostatni zapis wygrywa*. Jest ona powszechnie stosowana w bazach z replikacją z wieloma liderami i w bazach bez lidera takich jak Cassandra [53] i Riak [54] (zob. punkt „Ostatni zapis wygrywa (odrzuć jednocześnie zapisy)”). Niektóre implementacje generują znaczniki czasu po stronie klienta, a nie na serwerze. To jednak nie zmienia zasadniczego problemu z podejściem ostatni zapis wygrywa:

- Zapisy z bazy mogą nieoczekiwanie zniknąć. Węzeł z opóźnionym zegarem nie może do momentu upływu różnicy czasowej nadpisać wartości zapisanych wcześniej przez węzeł z szybszym zegarem [54, 55]. Ten scenariusz może powodować niezauważalne usunięcie dowolnej ilości danych bez zgłaszania błędu w aplikacji.
- Mechanizm ostatni zapis wygrywa nie rozróżnia szybko wprowadzanych zapisów sekwencyjnych (na rysunku 8.3 inkrementacja u klienta B jednoznacznie zachodzi *po* zapisie u klienta A) od rzeczywiście jednoczesnych zapisów (gdy żadna jednostka zapisująca nie jest świadoma działań innych jednostek). Potrzebne są dodatkowe mechanizmy śledzenia przyczynowości (np. wektory wersji), aby zapobiegać jej naruszeniu (zob. punkt „Wykrywanie jednoczesnych zapisów”).
- Możliwe, że dwa węzły niezależnie wygenerują zapisy z tym samym znacznikiem czasu — zwłaszcza gdy zegary mają rozdzielczość tylko na poziomie milisekund. Do rozwiązywania takich konfliktów potrzebna jest dodatkowa wartość (może nią być duża liczba losowa), jednak to podejście też może prowadzić do naruszenia przyczynowości.

Dlatego choć kusząca jest myśl o rozwiązywaniu konfliktów przez zachowanie „najnowszej” wartości i odrzucenie innych, ważne jest, by mieć świadomość, że definicja pojęcia „najnowsza” zależy od lokalnego zegara czasu rzeczywistego, który może działać niepoprawnie. Nawet gdy używane są zegary ściśle zsynchronizowane za pomocą serwerów NTP, możesz przesłać pakiet ze znacznikiem czasu 100 ms (według zegara nadawcy) i otrzymać go, gdy według zegara odbiorcy czas to 99 ms. Wydaje się wtedy, że pakiet dotarł jeszcze przed wysłaniem, co jest niemożliwe.

Czy synchronizacja NTP może być tak precyzyjna, by tego rodzaju nieprawidłowe uporządkowanie nie było możliwe? Zapewne nie, ponieważ precyzja synchronizacji NTP jest ograniczona przez czas przesyłu komunikatów w sieci i dodatkowe źródła błędów takie jak odchylenia zegara kwarcowego. Aby kolejność była prawidłowa, źródło czasu musi być znacznie precyzyjniejsze niż to, czego pomiar dotyczy (czyli opóźnienie sieciowe).

Bezpieczniejszym sposobem porządkowania zdarzeń są tzw. *zegary logiczne* [56, 57] oparte na zwiększanych licznikach, a nie na oscylatorach kwarcowych (zob. punkt „Wykrywanie jednoczesnych zapisów”). Zegary logiczne nie mierzą godziny ani liczby sekund, a jedynie względne uporządkowanie zdarzeń (czy jedno zdarzenie nastąpiło przed innym, czy po nim). Natomiast zegary dnia rzeczywistego i zegary monotoniczne mierzące rzeczywisty upływ czasu są nazywane *zegarami fizycznymi*. Więcej o porządkowaniu zdarzeń dowiesz się z punktu „Gwarancje związane z porządkowaniem”.

### Zakres pewności odczytów zegara

Zdarza się, że odczyt zegara czasu rzeczywistego z maszyny jest możliwy z rozdzielczością na poziomie mikro- lub nawet nanosekund. Jednak nawet jeśli zdołasz uzyskać tak szczegółowy pomiar, nie oznacza to, że jest on dokładny na takim poziomie. Zwykle jest inaczej. Wcześniej wspomniano, że odchylenie w nieprecyzyjnych zegarach kwarcowych może wynosić kilka milisekund nawet w sytuacji, gdy co sekundę synchronizujesz je z serwerem NTP w sieci lokalnej. Jeśli używasz serwera NTP w publicznym internecie, możesz uzyskać precyzję maksymalnie na poziomie dziesiątek milisekund, a gdy sieć jest przeciążona, błąd może szybko wzrosnąć do ponad 100 ms [57].

Dlatego traktowanie odczytów zegara jak punktu w czasie nie ma sensu. Te odczyty określają raczej zakresy czasu z określonymi zakresami pewności. Na przykład system może mieć 95% pewności, że czas wynosi obecnie od 10,3 do 10,5 s po minucie, jednak nie potrafi określić czasu z większą precyzją [58]. Jeśli znany jest czas z przybliżeniem  $\pm 100$  ms, mikrosekundy w znaczniku czasu są w praktyce bez znaczenia.

Poziom pewności można wyznaczyć na podstawie źródła czasu. Jeśli używasz podłączonego do komputera odbiornika GPC lub zegara atomowego (cezowego), oczekiwany zakres błędu jest podawany przez producenta. Jeżeli pobierasz czas z serwera, poziom błędu zależy od oczekiwanego odchylenia zegara kwarcowego od czasu ostatniej synchronizacji z serwerem, błędu serwera NTP i czasu wymiany komunikatów w sieci z serwerem (przybliżonego i przy założeniu, że ufasz serwerowi).

Niestety, większość systemów nie określa poziomu pewności. Na przykład po wywołaniu funkcji `clock_gettime()` zwracana wartość nie określa oczekiwanego błędu dla czasu. Nie wiesz więc, czy zakres pewności wynosi pięć milisekund czy pięć lat.

Ciekawym wyjątkiem jest interfejs API *TrueTime* Google’a w bazie Spanner [41], który bezpośrednio podaje zakres pewności dla lokalnego zegara. Gdy zażądasz aktualnego czasu, otrzymasz dwie wartości: *[najwcześniej, najpóźniej]* określające *najwcześniejszy możliwy i najpóźniejszy możliwy* czas. Na podstawie obliczeń poziomu pewności zegar stwierdza, że aktualny czas znajduje się w podanym przedziale. Szerokość przedziału zależy m.in. od tego, ile upłynęło od czasu ostatniej synchronizacji lokalnego zegara kwarcowego z precyzyjniejszym źródłem czasu.

## Synchronizowane zegary używane do tworzenia globalnych snapshotów

W punkcie „Izolacja snapshotów i powtarzalny odczyt” omówiono *izolację snapshotów*, która jest bardzo przydatną funkcją w bazach wymagających jednoczesnej obsługi małych, szybkich transakcji z odczytem i zapisem oraz dużych, długich transakcji z samym odczytem (np. na potrzeby analityki lub tworzenia kopii zapasowych). Izolacja snapshotów umożliwia transakcjom z samym odczytem zobaczenie bazy w spójnym stanie z określonego momentu bez tworzenia blokad i zakłócenia transakcji z odczytem i zapisem.

Najczęściej stosowana implementacja izolacji snapshotów wymaga monotonicznie rosnących identyfikatorów transakcji. Jeśli zapis nastąpi później niż wykonanie snapshota (czyli zapis ma wyższy identyfikator transakcji niż snapshot), zapis jest niewidoczny dla transakcji używających snapshota. W bazach działających w jednym węźle do generowania identyfikatorów transakcji wystarcza prosty licznik.

Jednak gdy baza jest rozproszona między wiele maszyn (potencjalnie z wielu centrów danych), trudno jest wygenerować globalny, monotonicznie rosnący identyfikator transakcji (dla wszystkich partycji), ponieważ wymaga to koordynacji. Identyfikator transakcji musi odzwierciedlać przyczynowość. Jeśli transakcja B wczytuje wartość zapisaną przez transakcję A, B musi mieć wyższy identyfikator transakcji niż A. W przeciwnym razie snapshot nie jest spójny. Gdy wykonywanych jest wiele małych, szybkich transakcji, generowanie identyfikatorów transakcji w systemie rozproszonym staje się nieakceptowalnym wąskim gardłem<sup>7</sup>.

Czy jako identyfikatorami transakcji można się posłużyć znacznikami czasu z synchronizowanych zegarów czasu rzeczywistego? Jeśli uda się zapewnić wystarczająco dobrą synchronizację, znaczniki czasu zyskają odpowiednie cechy — późniejsze transakcje będą miały wyższe znaczniki czasu. Problemem jest oczywiście brak pewności co do precyzji zegarów.

W bazie Spanner w ten sposób implementowana jest izolacja snapshotów między centrami danych [59, 60]. Używane są zakresy pewności zegara zwracane przez interfejs API TrueTime, a system działa zgodnie z następującą obserwacją: jeśli występują dwa zakresy pewności obejmujące najwcześniejszy i najpóźniejszy możliwy czas ( $A = [A_{\text{najwcześniejszy}}, A_{\text{najpóźniejszy}}]$  i  $B = [B_{\text{najwcześniejszy}}, B_{\text{najpóźniejszy}}]$ ) oraz te dwa przedziały się nie pokrywają (tzn.  $A_{\text{najwcześniejszy}} < A_{\text{najpóźniejszy}} < B_{\text{najwcześniejszy}} < B_{\text{najpóźniejszy}}$ ), to B z pewnością nastąpiło po A. Nie ma co do tego wątpliwości. Tylko wtedy, gdy przedziały nachodzą na siebie, nie wiadomo, jaka była kolejność A i B.

Aby mieć pewność, że znaczniki czasu transakcji odzwierciedlają przyczynowość, Spanner celowo odczekuje długość zakresu pewności przed zatwierdzeniem transakcji z odczytem i zapisem. W ten sposób gwarantuje, że każda transakcja, która może wczytać dane, będzie odpowiednio opóźniona, dzięki czemu zakresy pewności nie będą się pokrywać. Aby czas oczekiwania był możliwie krótki, Spanner musi utrzymywać niepewność co do czasu na jak najniższym poziomie. W tym celu Google w każdym centrum danych używa odbiornika GPS lub zegara atomowego, co pozwala na synchronizację zegarów z dokładnością do ok. 7 ms [41].

<sup>7</sup> Istnieją rozproszone generatory numerów porządkowych, np. Snowflake z serwisu Twitter, generujące w przybliżeniu monotonicznie rosnące unikatowe identyfikatory w skalowalny sposób (np. przydzielające różnym węzłom bloki z przestrzeni identyfikatorów). Jednak to podejście zwykle nie gwarantuje uporządkowania spójnego przyczynowo, ponieważ skala czasu, w jakiej przydzielane są bloki identyfikatorów, jest dłuższa niż skala czasu używana dla odczytów i zapisów w bazie. Zob. też „Gwarancje związane z porządkowaniem”.

Wykorzystywanie synchronizacji zegarów w funkcjonowaniu transakcji rozproszonych to obszar aktywnych badań [57, 61, 62]. Pomysły w tej dziedzinie są ciekawe, jednak poza rozwiązaniami Google’a nie zostały jeszcze zaimplementowane w popularnych bazach danych.

## Wstrzymywanie procesów

Zastanówmy się teraz nad innym przykładem niebezpiecznego zastosowania zegara w systemie rozproszonym. Załóżmy, że używasz bazy z jednym liderem na partycję. Tylko lider może akceptować zapisy. Skąd węzeł ma wiedzieć, że wciąż jest liderem (czyli nie został uznany za niesprawny przez inne jednostki) i że może bezpiecznie akceptować zapisy?

Jedną z możliwości jest uzyskiwanie przez lidera *dzierżawy* od innych węzłów. Dzierżawa działa jak blokada bez limitu czasu [63]. W danym momencie tylko jeden węzeł może utrzymywać dzierżawę. Dlatego gdy węzeł ją uzyska, wie, że przez pewien czas (do momentu wygaśnięcia dzierżawy) będzie liderem. Aby pozostać liderem, węzeł musi okresowo odnawiać dzierżawę przed jej wygaśnięciem. Jeśli węzeł ulegnie awarii, nie będzie mógł odnowić dzierżawy, dzięki czemu po jej wygaśnięciu inny węzeł przejmie funkcję lidera.

Możesz wyobrazić sobie następującą pętlę obsługi żądań:

```
while (true) {
    request = getIncomingRequest();

    // Upewnianie się, że dzierżawa jest ważna jeszcze przynajmniej 10 s
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

Jaki problem występuje w tym kodzie? Po pierwsze, kod polega na synchronizowanych zegarach. Czas wygasania dzierżawy jest ustawiany przez inną maszynę (gdzie wygasanie może być ustawiane np. jako bieżący czas plus 30 s), a porównywany z zegarem lokalnego systemu. Jeśli wskazania obu zegarów różnią się o więcej niż kilka sekund, ten kod może zacząć działać w dziwny sposób.

Po drugie, nawet jeśli zmodyfikujemy protokół w taki sposób, aby używał wyłącznie lokalnego zegara monotonicznego, pojawia się następny problem — kod przyjmuje, że od momentu sprawdzania czasu (`System.currentTimeMillis()`) do chwili przetwarzania żądania (`process(request)`) mija niewiele czasu. Ten kod zwykle działa bardzo szybko, dlatego bufor 10 s przeważnie wystarczy z zapasem do tego, by zagwarantować, iż dzierżawa nie wygaśnie w trakcie przetwarzania żądania.

Co się jednak stanie, gdy nastąpi nieoczekiwana przerwa w wykonywaniu programu? Wyobraź sobie, że wątek zatrzymuje się na 15 s w okolicy wiersza `lease.isValid()`, po czym kontynuuje pracę. Wtedy prawdopodobne jest, że dzierżawa wygaśnie przed przetworzeniem żądania i inny węzeł przejmie funkcję lidera. Jednak nic nie informuje wątku, że był wstrzymany przez tak długi czas. Dlatego kod zauważy wygaśnięcie dzierżawy dopiero w następnej iteracji pętli, jednak możliwe, że wykona już wtedy niebezpieczne operacje, przetwarzając żądanie.

Czy szaleństwem jest przyjmować, że wątek może zostać wstrzymany na tak długi czas? Niestety nie. Wstrzymanie pracy może wynikać z różnych powodów:

- Środowiska uruchomieniowe z wielu języków programowania (np. maszyna wirtualna Javy) używają *mechanizmu odzyskiwania pamięci*, który okresowo musi wstrzymać wszystkie działające wątki. Przerwy typu „zatrzymaj świat” trwają czasem kilka minut [64]! Nawet tzw. współbieżne mechanizmy odzyskiwania pamięci, np. CMS z maszyny wirtualnej HotSpot Javy, nie potrafią działać w pełni równolegle względem kodu aplikacji. Nawet one muszą od czasu do czasu „zatrzymać świat” [65]. Choć takie przerwy można często skrócić, zmieniając wzorce przydziału pamięci lub dostosowując ustawienia mechanizmu odzyskiwania pamięci [66], to jeśli chcesz zapewniać gwarancje niezawodności, musisz zakładać najgorszy scenariusz.
- W środowiskach z wirtualizacją praca maszyny wirtualnej może zostać *wstrzymana* (co powoduje wstrzymanie wykonywania wszystkich procesów i zapisanie zawartości pamięci na dysku) i *wznowiona* (co oznacza odzyskanie zawartości pamięci i kontynuowanie pracy). Przerwa może nastąpić w dowolnym momencie wykonywania procesu i trwać dowolnie długo. Ten mechanizm jest czasem używany do *migracji na żywo* maszyn wirtualnych z jednego hosta na inny bez restartu. Wtedy długość przerwy zależy od szybkości zapisu danych przez procesy w pamięci [67].
- Na urządzeniach użytkowników końcowych (np. w laptopach) wykonywanie kodu też może być dowolnie wstrzymywane i wznowiane — np. w momencie zamknięcia pokrywy laptopa.
- Gdy system operacyjny przełącza kontekst, aktywując inny wątek, lub gdy hiperwizor przełącza maszyny wirtualne (gdy kod działa w takiej maszynie), aktualnie działający wątek może zostać wstrzymany w dowolnym miejscu kodu. W maszynie wirtualnej czas, jaki procesor przeznacza dla innych maszyn wirtualnych, jest nazywany *kradzionym* (ang. *steal time*). Jeśli maszyna jest mocno obciążona (czyli gdy kolejka wątków oczekujących na uruchomienie jest długa), do momentu wznowienia pracy przez wstrzymany wątek może minąć dużo czasu.
- Jeśli aplikacja synchronicznie korzysta z dysku, wątek może zostać wstrzymany w oczekiwaniu na zakończenie powolnej dyskowej operacji wejścia-wyjścia [68]. W wielu językach dostęp do dysku może następować w zaskakujących momentach — nawet gdy w kodzie nie ma bezpośredniego dostępu do dysku. Na przykład mechanizm wczytywania klas Javy leniwie wczytuje pliki klas w momencie ich pierwszego użycia. Może to nastąpić w dowolnym punkcie wykonywania programu. Przerwy spowodowane operacjami wejścia-wyjścia i mechanizmem odzyskiwania pamięci mogą się nawet łączyć [69]. Jeśli dysk jest sieciowym systemem plików lub sieciowym urządzeniem blokowym (takim jak EBS Amazonu), opóźnienie operacji wejścia-wyjścia jest dodatkowo nasilane przez zmienność opóźnienia w sieci [29].
- Gdy konfiguracja systemu operacyjnego dopuszcza *stronicowanie*, prosty dostęp do pamięci może skutkować błędem stronicowania, co wymaga wczytania strony z dysku do pamięci. Wątek jest wtedy wstrzymywany na czas wykonywania tej powolnej operacji wejścia-wyjścia. Jeśli wykorzystanie pamięci jest wysokie, może to prowadzić do zapisania na dysku innej strony. W skrajnych sytuacjach system operacyjny większość czasu poświęca na przenoszenie stron między pamięcią a dyskiem, przez co wykonuje niewiele rzeczywistej pracy (taki problem to *szamotanie*; ang. *thrashing*). Aby uniknąć tego problemu, na serwerach stronicowanie jest często wyłączane (jeśli operator woli zamknąć proces w celu zwolnienia pamięci, niż ryzykować szamotanie).



- Proces uniksowy można wstrzymać, przesyłając sygnał SIGSTOP (np. w wyniku wciśnięcia kombinacji *Ctrl+Z* w powłoce). Ten sygnał natychmiast uniemożliwia procesowi wykorzystanie dalszych cykli procesora do czasu wznowienia pracy za pomocą sygnału SIGCONT. Wtedy proces wznowia działanie od miejsca, w którym został wstrzymany. Nawet jeśli Twoje środowisko standardowo nie używa sygnału SIGSTOP, może on zostać przypadkowo przesłany przez inżyniera odpowiedzialnego za eksploatację.

Wszystkie opisane sytuacje mogą spowodować *wyłączenie* działającego wątku w dowolnym momencie i późniejsze wznowienie go, a wątek nawet tego nie zauważy. Ten problem jest podobny do zapewniania wielowątkowemu kodowi bezpieczeństwa ze względu na wątki na jednej maszynie. Nie można wtedy przyjmować żadnych założeń co do czasu, ponieważ możliwe są współbieżność i arbitralne przypadki przełączania kontekstu.

Na potrzeby pisania kodu wielowątkowego do wykonywania na jednej maszynie dostępne są stosunkowo dobre narzędzia pomagające zapewnić bezpieczeństwo ze względu na wątki. Te narzędzia to: muteksy, semaforey, liczniki atomowe, struktury danych niewymagające blokad, kolejki z blokadami itd. Niestety, te narzędzia nie przekładają się bezpośrednio na systemy rozproszone, ponieważ w takich systemach nie ma pamięci współużytkowanej — występują tylko komunikaty przesyłane przez zawodną sieć.

Węzeł w systemie rozproszonym musi zakładać, że jego praca może zostać wstrzymana na dłuższy czas w dowolnym momencie, nawet w połowie wykonywania funkcji. W czasie przerwy reszta świata wciąż działa i może nawet uznać wstrzymany węzeł za niesprawny, ponieważ nie reaguje. Ostatecznie wstrzymany węzeł może wznowić pracę, nawet nie zauważając, że był uśpiony (wykrywa to dopiero po późniejszym sprawdzeniu zegara).

## Gwarancje czasu zwracania odpowiedzi

W wielu językach programowania i systemach operacyjnych wątki oraz procesy mogą zostać wstrzymane na nieograniczony czas. Wkładając w to odpowiednio dużo wysiłku, *można* wyeliminować przyczyny takich przerw.

Czasem oprogramowanie działa w środowiskach, gdzie brak odpowiedzi w określonym czasie może spowodować poważne szkody. Komputery kontrolujące samoloty, rakiety, roboty, samochody i inne obiekty fizyczne muszą reagować szybko i przewidywalnie na dane wejściowe z czujników. W takich systemach występuje określony *limit czasu*, w jakim oprogramowanie musi odpowiedzieć. Jeśli program nie zmieści się w tym limicie, może spowodować awarię całego systemu. Są to tzw. systemy o *ostrych ograniczeniach czasowych* (ang. *hard real-time*).



### Czy czas rzeczywisty naprawdę jest rzeczywisty?

W systemach wbudowanych *czas rzeczywisty* oznacza, że system został starannie zaprojektowany i przetestowany, aby we wszystkich okolicznościach spełniał określone gwarancje związane z czasem. Oznacza to coś innego niż bardziej wieloznaczne określenie *czas rzeczywisty* w kontekście internetu, określające serwery przesyłające dane do klientów i przetwarzanie strumieniowe bez ostrych ograniczeń czasu reakcji (zob. rozdział 11.).

Nie chcesz np., aby po wykryciu wypadku przez czujniki samochodu otwarcie poduszki powietrznej zostało opóźnione z przyczyny spowodowanej przez mechanizm odzyskiwania pamięci niefortunnej przerwy w systemie uruchamiania poduszki.

Zapewnianie w systemie gwarancji na poziomie czasu rzeczywistego wymaga obsługi na wszystkich poziomach stosu oprogramowania. Potrzebny jest *system operacyjny czasu rzeczywistego* umożliwiający szeregowanie procesów z gwarantowanym przydziałem czasu procesora w ustalonych okresach. Dla funkcji bibliotecznych trzeba udokumentować czas wykonania w najgorszym przypadku. Dynamiczny przydział pamięci trzeba ograniczyć lub całkowicie wyeliminować (istnieją mechanizmy odzyskiwania pamięci w czasie rzeczywistym, jednak aplikacja nadal musi zapewniać, że nie będzie wymagać od takich mechanizmów zbyt dużo pracy). Ponadto konieczne są bardzo rozbudowane testy i pomiary, aby się upewnić, że gwarancje są dotrzymywane.

Wszystko to wymaga dużej ilości dodatkowej pracy i znacznie ogranicza zakres języków programowania, bibliotek i narzędzi, jakie można stosować (ponieważ większość języków i narzędzi nie zapewnia gwarancji na poziomie czasu rzeczywistego). Z tych powodów budowanie systemów czasu rzeczywistego jest bardzo kosztowne. Takie systemy najczęściej występują w urządzeniach wbudowanych kluczowych ze względu na bezpieczeństwo. Ponadto „czas rzeczywisty” nie oznacza tego samego co „wysoka wydajność”. W praktyce systemy czasu rzeczywistego mogą mieć niższą przepustowość, ponieważ muszą priorytetowo traktować czas odpowiedzi (zob. też punkt „Opóźnienie i wykorzystanie zasobów”).

W większości systemów przetwarzania danych działających po stronie serwera gwarancje na poziomie czasu rzeczywistego są nieekonomiczne i nieodpowiednie. Dlatego te systemy są narażone na przerwy i niestabilność zegarów wynikające z działania w środowisku, które nie pracuje w czasie rzeczywistym.

### Ograniczanie skutków odzyskiwania pamięci

Negatywne skutki wstrzymywania procesu można złagodzić bez uciekania się do kosztownych gwarancji szeregowania w czasie rzeczywistym. Środowiska uruchomieniowe języków zapewniają pewną swobodę w zakresie odzyskiwania pamięci, ponieważ śledzą przydział zasobów dla obiektów i ilość wolnej pamięci.

Nowy pomysł polega na traktowaniu przerw spowodowanych odzyskiwaniem pamięci jak krótkich planowanych przestojów węzła. Gdy jeden węzeł odzyskuje pamięć, system pozwala innym węzłom na obsługę żądań od klientów. Jeśli środowisko uruchomieniowe potrafi ostrzegać aplikację, że węzeł wkrótce będzie wymagał przerwy spowodowanej odzyskiwaniem pamięci, aplikacja może wstrzymać przesyłanie do tego węzła nowych żądań i poczekać na zakończenie przetwarzania w nim zaległych żądań. Następnie węzeł może uruchomić odzyskiwanie pamięci w czasie, gdy żadne żądania nie są przetwarzane. Ta sztuczka ukrywa przed klientami spowodowane odzyskiwaniem pamięci przerwy i skraca czas odpowiedzi dla wysoki percentyli [70, 71]. To podejście jest stosowane w niektórych wrażliwych na opóźnienia finansowych systemach tradingowych [72].

Inną wersją tego pomysłu jest odzyskiwanie pamięci tylko dla krótkotrwałych obiektów (co można zrobić szybko) i okresowe restartowanie procesów, zanim zakumulują na tyle dużo długotrwałych obiektów, aby konieczne było pełne odzyskiwanie pamięci obejmujące takie obiekty [65, 73]. Można restartować za każdym razem po jednym węźle i przenosić ruch z węzła przed planowanym restartem (tak jak w stopniowej aktualizacji; zob. rozdział 4.).

Te techniki nie zapobiegają całkowicie przerwom powodowanym odzyskiwaniem pamięci, ale mogą w przydatny sposób ograniczyć wpływ takich przerw na aplikację.

# Wiedza, prawda i kłamstwa

Do tej pory analizowaliśmy w tym rozdziale aspekty, w których systemy rozproszone różnią się od programów działających w jednym komputerze. W systemach rozproszonych nie występuje współużytkowana pamięć, a jedynie komunikaty przekazywane za pomocą zawodnej sieci ze zmiennym opóźnieniem. Systemy rozproszone są też narażone na częściowe awarie, zawodne zegary i przerwy w przetwarzaniu.

Skutki tych problemów są wysoce dezorientujące, jeśli nie jesteś przyzwyczajony do systemów rozproszonych. Węzeł w sieci nie może niczego *wiedzieć* z pewnością. Może tylko zgadywać na podstawie komunikatów otrzymanych (lub nieotrzymanych) za pomocą sieci. Węzeł może ustalić stan innego węzła (jakie dane przechowuje, czy działa poprawnie itd.) wyłącznie poprzez wymianę komunikatów. Jeśli zdalny węzeł nie odpowiada, nie wiadomo, w jakim znajduje się stanie, ponieważ problemów z siecią nie da się wiarygodnie odróżnić od problemów z węzłem.

Dyskusje na temat takich systemów ocierają się o kwestie filozoficzne — jaką mamy wiedzę o tym, co jest prawdziwe, a co fałszywe w danym systemie? W jakim stopniu możemy być pewni tej wiedzy, jeśli mechanizmy służące do obserwacji i pomiarów są zawodne? Czy systemy oprogramowania powinny być zgodne z prawami, jakich oczekujemy od fizycznego świata, np. z prawem przyczyny i skutku?

Na szczęście nie musimy posuwać się do szukania sensu życia. W systemie rozproszonym można określić założenia na temat jego zachowania (*model systemu*) i zaprojektować systemy w taki sposób, aby spełniał te założenia. Można udowodnić poprawność pracy algorytmów w określonym modelu systemu. To oznacza, że niezawodne działanie jest osiągalne, nawet jeśli dany model systemu zapewnia bardzo nieliczne gwarancje.

Jednak choć możliwe jest zapewnienie sprawnej pracy oprogramowania na podstawie zawodnego modelu systemu, nie jest to łatwe. W dalszej części rozdziału dokładniej omówione są zagadnienia wiedzy i prawdy w systemach rozproszonych. Pomoże Ci to zastanowić się nad założeniami, jakie można przyjmować, i gwarancjami, które możesz chcieć zapewniać. W rozdziale 9. znajdziesz przykłady systemów rozproszonych i algorytmy zapewniające określone gwarancje na podstawie konkretnych założeń.

## Prawda jest definiowana przez większość

Wyobraź sobie sieć z asymetrycznym błędem, w której węzeł otrzymuje wszystkie przesłane do niego komunikaty, ale komunikaty wychodzące z tego węzła są gubione lub opóźnione [19]. Choć ten węzeł działa w pełni poprawnie i otrzymuje żądania od innych węzłów, pozostałe węzły nie otrzymują od niego odpowiedzi. Po przekroczeniu limitu czasu inne węzły uznają dany węzeł za niesprawny, ponieważ nie otrzymały od niego informacji. Ta sytuacja przypomina koszmar. Częściowo odłączony węzeł zostaje wrzucony do grobu, gdzie wierzga i krzyczy: „Nie jestem martwy!”. Jednak ponieważ nikt go nie słyszy, kondukt pogrzebowy ze stoickim spokojem idzie dalej.

W nieco mniej koszmarnym scenariuszu częściowo odłączony węzeł może wykryć, że inne węzły nie potwierdzają otrzymanych wysyłanych przez niego komunikatów. Stwierdza więc, że występuje błąd w sieci. Mimo to częściowo odłączony węzeł zostaje niesłusznie uznany za niesprawny przez inne węzły i nie może nic z tym zrobić.

Oto trzeci scenariusz: wyobraź sobie węzeł, w którym występuje długa przerwa typu „zatrzymać świat” na odzyskiwanie pamięci. Wszystkie wątki w węźle są wywłaszczane przez mechanizm odzyskiwania pamięci i wstrzymywane na minutę. Dlatego węzeł nie przetwarza żądań i nie wysyła odpowiedzi. Inne węzły oczekują na odpowiedź, ponawiają próbę, tracą cierpliwość i ostatecznie uznają węzeł za niesprawny, po czym umieszczają go na katafalku. Ostatecznie mechanizm odzyskiwania pamięci kończy pracę i wątki węzła kontynuują działania tak, jakby nic się nie stało. Pozostałe węzły są zaskoczone, gdy uznany za martwy węzeł nagle podnosi głowę z trumny i w pełni zdrowia zaczyna radośnie rozmawiać z przechodniami. Początkowo węzeł, w którym następuje odzyskiwanie pamięci, nie zdaje sobie nawet sprawy z tego, że upłynęła cała minuta i że został uznany za martwy. Z jego perspektywy od czasu ostatniej komunikacji z innymi węzłami czas prawie nie upłynął.

Morał z tych historii jest taki, że węzeł nie zawsze może ufać swojej ocenie sytuacji. System rozproszony nie może całkowicie polegać na jednym węźle, ponieważ może on w dowolnym momencie zawieść, co może skutkować zablokowaniem systemu i niemożnością przywrócenia stanu. Zamiast tego wiele algorytmów rozproszonych polega na *kworum*, czyli na głosowaniu z udziałem węzłów (zob. punkt „Odczyty i zapisy zgodne z kworum”). Decyzje wymagają minimalnej liczby głosów z kilku węzłów, co zmniejsza zależność od pojedynczych węzłów.

Dotyczy to także decyzji o uznaniu węzłów za niesprawne. Jeśli kworum zadecyduje, że inny węzeł jest niesprawny, musi on zostać uznany za „martwy”, nawet gdy wciąż czuje się pełen życia. Pojedynczy węzeł musi zaakceptować decyzję kworum i zrezygnować.

Za kworum najczęściej uznaje się bezwzględną większość ponad połowy węzłów (choć możliwe są też inne rodzaje kworum). Kworum większościowe umożliwia systemowi kontynuowanie pracy po awarii pojedynczych węzłów (przy trzech węzłach tolerowana jest jedna awaria; przy pięciu węzłach dopuszczalne są dwie awarie). Jednak to rozwiązanie jest bezpieczne, ponieważ w systemie może występować tylko jedna większość. Nie może być jednocześnie dwóch większości podejmujących sprzeczne decyzje. Posługiwanie się kworum jest szczegółowo opisane w kontekście *algorytmów uzgadniania konsensusu* w rozdziale 9.

## Lider i blokady

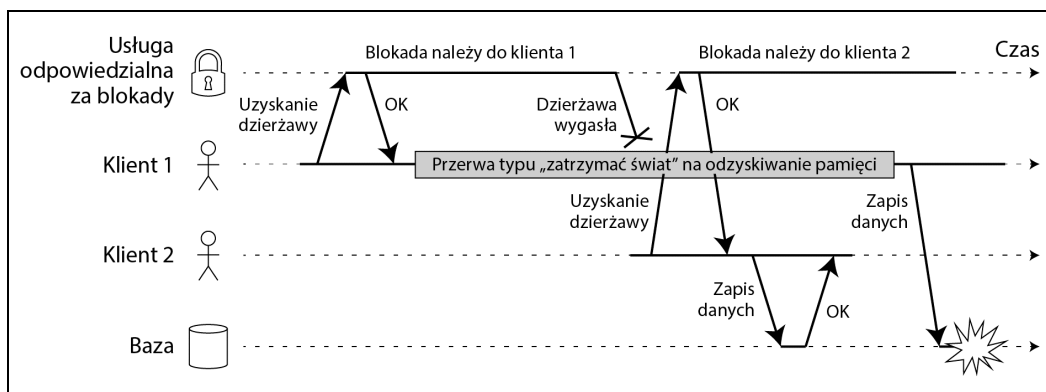
System często wymaga, by istniał tylko jeden egzemplarz jakiejś rzeczy. Oto przykłady:

- Tylko jeden węzeł może być liderem partycji bazy danych, co pozwala uniknąć „rozszczepienia mózgu” (zob. punkt „Radzenie sobie z przestojami węzłów”).
- Tylko jedna transakcja lub jeden klient mogą zajmować blokadę określonego zasobu lub obiektu, aby zapobiec jednoczesnemu zapisowi danych i uszkodzeniu tego elementu.
- Tylko jeden użytkownik może zarejestrować konkretną nazwę, ponieważ musi ona jednoznacznie identyfikować daną osobę.

Implementacja takich rozwiązań w systemie rozproszonym wymaga ostrożności. Nawet jeśli węzeł sądzi, że jest „wybrańcem” (liderem partycji, właścicielem blokady, funkcją obsługi żądania od użytkownika, który z powodzeniem zajął daną nazwę), nie musi to oznaczać, że kworum węzłów się z tym zgadza! Węzeł mógł być wcześniej liderem, jeśli jednak inne węzły w międzyczasie uznały, że jest niesprawny (np. z powodu zakłóceń w sieci lub przerwy spowodowanej odzyskiwaniem pamięci), mógł zostać zastąpiony przez wybranego innego lidera.

Jeśli węzeł wciąż działa jak „wybraniec”, choć większość węzłów uznała go za niesprawny, może to spowodować problemy w systemie (jeżeli nie został on starannie zaprojektowany). Taki węzeł może przysyłać do innych komunikaty zgodnie ze swoimi samowolnymi uprawnieniami, a jeśli inne węzły w nie uwierzą, system jako całość może wykonywać nieprawidłowe operacje.

Na przykład na rysunku 8.4 widoczne jest uszkodzenie danych spowodowane nieprawidłową implementacją blokad. Ten błąd nie jest czysto teoretyczny — taki problem występował w bazie HBase [74, 75]. Załóżmy, że chcesz się upewnić, iż w usłudze bazodanowej dostęp do pliku może uzyskać tylko jeden klient naraz (ponieważ próba zapisu przez wiele klientów spowoduje uszkodzenie pliku). Próbujesz zaimplementować to rozwiązanie, wymagając, by klient przed dostępem do pliku uzyskał dzierżawę od usługi odpowiedzialnej za blokady.



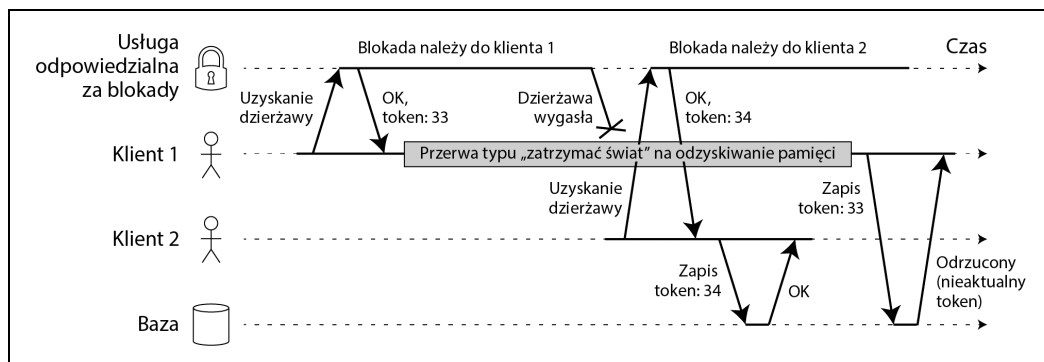
Rysunek 8.4. Nieprawidłowa implementacja blokady rozproszonej. Klient 1 uważa, że nadal ma aktualną dzierżawę, choć zdążyła już ona wygasnąć. Skutkuje to uszkodzeniem pliku w bazie

Ten problem to przykład sytuacji opisanej w punkcie „Wstrzymywanie procesów”. Jeśli klient posiadający dzierżawę zostaje wstrzymany na zbyt długi czas, dzierżawa wygasa. Inny klient może wtedy uzyskać dzierżawę tego samego pliku i zacząć zapisywać w nim dane. Gdy wstrzymany klient wznowia działanie, zakłada (niesłusznie), że nadal ma aktywną dzierżawę, i także zapisuje dane w pliku. W efekcie zapisy klientów powodują konflikt i uszkadzają plik.

### Tokeny odgradzające

Gdy stosujesz blokadę lub dzierżawę do zabezpieczania dostępu do jakiegoś zasobu (np. pliku z danymi na rysunku 8.4), musisz się upewnić, że węzeł działający z błędnym założeniem, iż jest „wybrańcem”, nie zakłóci pracy reszty systemu. Stosunkowo prostą techniką pozwalającą osiągnąć ten cel jest przedstawione na rysunku 8.5 *odgradzanie* (ang. *fencing*).

Założmy, że odpowiedzialny za blokady serwer za każdym razem, gdy przyznaje blokadę lub dzierżawę, zwraca też *token odgradzający*, którym jest liczba rosnąca przy każdym przyznaniu blokady (ta wartość jest zwiększana np. przez usługę obsługującą blokady). Następnie można wymagać, by klient za każdym razem, gdy przesyła do usługi bazodanowej żądanie zapisu, dołączał też aktualny token odgradzający.



Rysunek 8.5. Zabezpieczanie dostępu do bazy dzięki zezwalaniu na zapisy tylko zgodnie z porządkiem rosnących tokenów odgradzających

Na rysunku 8.5 klient 1 uzyskuje dzierżawę razem z tokenem 33, jednak później zostaje wstrzymany na długi czas i dzierżawa wygasa. Klient 2 uzyskuje dzierżawę razem z tokenem 34 (wartości tokenów zawsze rosną), a następnie przesyła do usługi bazodanowej żądanie zapisu i token 34. Potem klient 1 wznowia pracę i przesyła do usługi bazodanowej zapis i token 33. Jednak serwer bazodanowy pamięta, że przetwarzał już zapis z większym tokenem (34), dlatego odrzuca żądanie z tokenem 33.

Jeśli jako usługa odpowiedzialna za blokady używany jest ZooKeeper, tokenem odgradzającym może być identyfikator transakcji (zxid) lub wersja węzła (cversion). Ponieważ istnieje gwarancja, że te wartości są monotonicznie rosnące, mają one pożądane cechy [74].

Zauważ, że ten mechanizm wymaga, by sam zasób aktywnie sprawdzał tokeny i odrzucał zapisy z tokenami starszymi niż te, które już były przetwarzane. Nie wystarczy polegać na tym, że klienci same będą sprawdzały status blokad. Jeśli zasoby nie obsługują bezpośrednio tokenów odgradzających, czasem można poradzić sobie z tym problemem (np. w usłudze przechowującej dane w plikach możesz umieszczać taki token w nazwie pliku). Jednak jakiś test jest niezbędny, aby uniknąć przetwarzania żądań, gdy operacja nie jest chroniona blokadą.

Sprawdzanie tokenu po stronie serwera może wydawać się wadą, jednak można to także uznać za korzystne rozwiązanie. Nierozsądne jest zakładanie w usłudze, że klienci zawsze będą działać prawidłowo. Klienci są bowiem często używane przez osoby, których priorytety bardzo się różnią od celów właścicieli usługi [76]. Dlatego dobrym pomysłem jest, by usługa chroniła samą siebie przed klientami, które niewłaściwie wykorzystują dane im możliwości.

## Błędy bizantyjskie

Tokeny odgradzające pozwalają wykrywać i blokować węzły, które *przypadkowo* działają w nieprawidłowy sposób (np. dlatego, że nie odkryły jeszcze, iż ich dzierżawa wygasa). Jeśli jednak węzeł chce celowo naruszyć gwarancje systemu, może łatwo to zrobić, przysyłając komunikaty z fałszywymi tokenami.

W tej książce zakładamy, że węzły są zawodne, ale „uczciwe”. Mogą działać powoli lub nigdy nie odpowiadać (z powodu błędu), a ich stan może być nieaktualny (przez przerwę spowodowaną odzyskiwaniem pamięci lub przez opóźnienie sieciowe), jednak przyjmujemy, że jeśli węzeł *odpowiada*, mówi „prawdę” — zgodnie z jego najlepszą wiedzą i regułami protokołu.

## Problem generałów bizantyjskich

Problem generałów bizantyjskich to uogólnienie *problemu dwóch generałów* [78] dotyczącego sytuacji, w której dwóch generałów armii musi uzgodnić plan bitwy. Ponieważ generałowie stacjonują w różnych miejscach, mogą się komunikować tylko przez posłańca, który może się spóźnić lub zgubić (tak jak pakiety w sieci). Problem osiągnięcia *konsensusu* opisano w rozdziale 9.

W bizantyjskiej wersji problemu uzgodnić działania musi  $n$  generałów, a ich poczynania są utrudnione przez to, że w ich szeregach mogą się znajdować zdrajcy. Większość generałów jest lojalna i przesyła godne zaufania komunikaty. Jednak zdrajcy próbują oszukać i zmylić pozostałych generałów, przysyłając sfałszowane lub nieprawdziwe komunikaty, i jednocześnie starają się uniknąć wykrycia. Początkowo nie wiadomo, kto jest zdrajcą.

Bizancjum było starożytnym greckim miastem przemianowanym później na Konstantynopol. Leżało w miejscu obecnego Stambułu w Turcji. Nie istnieją historyczne dowody na to, że generałowie w Bizancjum byli bardziej skłonni do intryg i knowań niż dowódcy w innych regionach. Nazwa problemu pochodzi od określenia *bizantyjskie* oznaczającego *nadmiernie skomplikowane, biurokratyczne, zwodnicze* i stosowanego w polityce na długo przed epoką komputerów [79]. Lamport chciał zmienić nadoświadczenie generałów, aby nie sprawiać przykrości czytelnikom, jednak zasugerowano mu, że nazwa *problem albańskich generałów* nie jest najlepszym pomysłem [80].

Problemy w systemach rozproszonych stają się znacznie poważniejsze, jeśli istnieje ryzyko, że węzły mogą „kłamać” (wysłać dowolnie błędne lub uszkodzone odpowiedzi) — np. gdy węzeł może twierdzić, że otrzymał określony komunikat, podczas gdy w rzeczywistości wcale tak nie było. Takie działanie nazywa się *błędem bizantyjskim*, a zadanie osiągnięcia konsensusu w niezaufanym środowisku to *problem generałów bizantyjskich* [77].

System jest *odporny na błędy bizantyjskie*, jeśli działa poprawnie nawet wtedy, gdy niektóre węzły działają nieprawidłowo i nie są zgodne z protokołem lub gdy napastnicy zakłócają działanie sieci. To zagadnienie jest istotne w pewnych konkretnych warunkach. Oto przykłady:

- W środowiskach używanych w przestrzeni powietrznej dane w pamięci komputera lub w rejestrach procesora mogą zostać uszkodzone przez promieniowanie, przez co system może reagować na działania innych węzłów w nieprzewidywalny sposób. Ponieważ awaria takiego systemu byłaby bardzo kosztowna (np. samolot mógłby się rozbić i spowodować śmierć wszystkich pasażerów lub rakieta mogłaby trafić w Międzynarodową Stację Kosmiczną), systemy kontroli lotu muszą być odporne na błędy bizantyjskie [81, 82].
- W systemie obejmującym wiele jednostek niektóre z nich mogą próbować oszukać lub okraść inne. W takim scenariuszu dla węzła nie jest bezpiecznie ufać komunikatom od innym węzłów, ponieważ dane mogą być przesyłane w złym celu. Na przykład sieci P2P takie jak bitcoin lub inne systemy z łańcuchem bloków można uznać za sposób na uzgadnianie przez nieufające sobie strony tego, czy doszło do transakcji, czy nie, przy czym nie wymaga to polegania na jednostce centralnej [83].

Jednak w przypadku systemów omawianych w tej książce zwykle można bezpiecznie przyjąć, że nie występują w nich błędy bizantyjskie. W centrum danych wszystkie węzły są kontrolowane przez organizację (dlatego — miejmy nadzieję — można im ufać), a poziom promieniowania jest tak niski, że uszkodzenie pamięci nie stanowi poważnego problemu. Protokoły zapewniania odporności

systemu na błędy bizantyjskie są dość skomplikowane [84], a systemy wbudowane odporne na błędy korzystają z mechanizmów sprzętowych [81]. W większości systemów danych działających po stronie serwera koszt wdrażania mechanizmów odpornych na błędy bizantyjskie powoduje, że takie rozwiązania są niepraktyczne.

W aplikacjach internetowych trzeba oczekiwać arbitralnych i złośliwych zachowań klientów (np. przeglądarek internetowych) kontrolowanych przez użytkowników końcowych. To dlatego sprawdzanie poprawności i oczyszczanie danych wejściowych oraz sekwencje ucieczki w danych wyjściowych są tak ważne — np. w celu zapobiegania wstrzykiwaniu kodu w SQL-u i atakom typu cross-site scripting. Jednak zwykle nie stosuje się do tego protokołów odpornych na błędy bizantyjskie. W zamian serwer jest uznawany za autorytet w kwestii decydowania, które zachowania klienta są dozwolone, a które nie. W sieciach P2P, gdzie nie ma tego rodzaju jednostki centralnej, odporność na błędy bizantyjskie ma większe znaczenie.

Błąd w oprogramowaniu można uznać za błąd bizantyjski, jeśli jednak instalujesz to samo oprogramowanie we wszystkich węzłach, algorytm odporny na błędy bizantyjskie Cię nie ochroni. Większość takich algorytmów do poprawnego działania wymaga większości kwalifikowanej ponad 2/3 węzłów (np. jeśli używasz czterech węzłów, najwyżej jeden może działać nieprawidłowo). Aby zastosować to podejście do ochrony przed błędami, potrzebowałbyś czterech niezależnych implementacji tego samego oprogramowania z nadzieją, że błąd występuje tylko w jednej z nich.

Atrakcyjnym rozwiązaniem byłby też protokół chroniący przed lukami, naruszeniami zabezpieczeń i złośliwymi atakami. Niestety, także to podejście jest nierealistyczne. W większości systemów jest tak, że jeśli napastnik może się włamać do jednego węzła, prawdopodobnie może to zrobić także z wszystkimi, ponieważ zapewne działa w nich to samo oprogramowanie. Dlatego głównym zabezpieczeniem przed napastnikami nadal są tradycyjne mechanizmy (uwierzytelnianie, kontrola dostępu, szyfrowanie, zapory itd.).

## Słabe formy kłamania

Choć zakładamy, że węzły są zwykle uczciwe, przydatne może być dodanie do oprogramowania mechanizmów chroniących przed słabymi formami „kłamania” — np. nieprawidłowymi komunikatami spowodowanymi problemami sprzętowymi, błędami oprogramowania i niepoprawną konfiguracją. Takie mechanizmy zabezpieczające nie zapewniają kompletnej odporności na błędy bizantyjskie, ponieważ nie powstrzymają zdeterminowanego napastnika. Są jednak prostym i praktycznym krokiem w kierunku wyższej niezawodności. Oto przykłady:

- Pakiety sieciowe zostają czasem uszkodzone z powodu problemów sprzętowych lub błędów w systemach operacyjnych, sterownikach, routerach itd. Uszkodzone pakiety są zwykle wykrywane za pomocą sum kontrolnych wbudowanych w protokoły TCP i UDP, jednak czasem pozostają niewykryte [85, 86, 87]. Zwykle wystarczająca ochronę przed takimi uszkodzeniami zapewniają proste rozwiązania takie jak sumy kontrolne w protokole na poziomie aplikacji.
- Publicznie dostępne aplikacje muszą starannie oczyszczać wszystkie dane wejściowe od użytkowników, np. sprawdzając, czy wartość znajduje się w akceptowalnym zakresie, i ograniczając długość łańcuchów znaków w celu ochrony przed atakami DoS spowodowanymi przydzieleniem dużej ilości pamięci. W wewnętrznej usłudze ukrytej za zaporą wystarczające mogą być mniej restrykcyjne testy danych wejściowych, warto jednak przeprowadzać proste sprawdzanie wartości (np. w protokole parsowania danych [85]).



- Klienci NTP można skonfigurować przy użyciu adresów wielu serwerów. W trakcie synchronizacji klient kontaktuje się wtedy z wszystkimi serwerami, szacuje błędy i sprawdza, czy większość serwerów zgodnie zwraca czas z określonego przedziału. Dopóki większość serwerów działa poprawnie, błędnie skonfigurowany serwer NTP zgłaszający nieprawidłowy czas jest uznawany za odstający od innych i pomijany w trakcie synchronizacji [37]. Wykorzystanie wielu serwerów sprawia, że protokół NTP działa znacznie stabilniej niż przy korzystaniu z tylko jednego serwera.

## Model systemu a rzeczywistość

Zaprojektowano wiele algorytmów do rozwiązywania problemów z systemami rozproszonymi (np. w rozdziale 9. omówiono rozwiązania problemu osiągnięcia konsensusu). Aby takie algorytmy były przydatne, muszą być odporne na różne opisane w tym rozdziale błędy typowe dla systemów rozproszonych.

Algorytmy trzeba pisać w taki sposób, aby nie były przesadnie zależne od szczegółów konfiguracji sprzętowej i programowej, przy użyciu której są uruchamiane. To wymaga sformalizowania rodzajów błędów, jakie mogą wystąpić w systemie. Służy do tego definicja *modelu systemu*, która jest abstrakcyjnym opisem założeń, jakie można przyjąć w algorytmie.

W kontekście założeń związanych z synchronizacją często używane są trzy modele systemów:

### *Model synchroniczny*

W modelu synchronicznym przyjmuje się ograniczone opóźnienie sieciowe, ograniczone przerwy procesów i ograniczony błąd zegara. Nie oznacza to w pełni zsynchronizowanych zegarów lub zerowego opóźnienia sieciowego. Wiadomo jednak, że opóźnienie sieciowe, przerwy i odchylenie zegara nigdy nie przekroczą ustalonego ograniczenia [88]. W większości stosowanych w praktyce systemów model synchroniczny jest nierealistyczny, ponieważ (co opisano w tym rozdziale) zdarzają się nieograniczone opóźnienia i przerwy.

### *Model częściowo synchroniczny*

Częściowa synchronizacja oznacza, że system *przez większość czasu* działa jak synchroniczny, jednak zdarza się przekroczenie ograniczeń opóźnienia sieciowego, przerw procesów i odchylenia zegara [88]. W wielu systemach jest to realistyczny model. Przez większość czasu sieci i procesy zachowują się poprawnie (w przeciwnym razie nie mogłyby wykonać żadnych zadań), trzeba jednak uwzględnić to, że założenia związane z synchronizacją okresowo okazują się nieprawdziwe. W takich sytuacjach opóźnienie sieciowe, przerwy i błędy zegara mogą być dowolnie duże.

### *Model asynchroniczny*

W tym modelu w algorytmie nie można przyjmować żadnych założeń związanych z synchronizacją. Co więcej, zegar w ogóle nie jest tu uwzględniany (dlatego nie można stosować limitów czasu). Niektóre algorytmy można zaprojektować z myślą o modelu asynchronicznym, jednak powoduje on poważne ograniczenia.

Obok synchronizacji trzeba uwzględnić też awarie węzłów. Trzy najczęściej stosowane modele systemów dotyczące węzłów to:

#### *Błędy typu awaria — przerwanie pracy*

W tym modelu algorytm przyjmuje, że możliwy jest tylko jeden rodzaj niepowodzeń — awarie. To oznacza, że węzeł może w dowolnym momencie przestać reagować, a następnie zostaje stale wyłączony (nigdy nie wznowia pracy).

#### *Błędy typu awaria — przywrócenie stanu*

Tu zakładamy, że węzły mogą w dowolnym momencie ulec awarii, przy czym po nieokreślonym czasie mogą ponownie zacząć reagować. W tym modelu przyjmujemy, że węzły mają stabilną pamięć (trwałą pamięć dyskową) zachowywaną między awariami, ale stan przechowywany w pamięci uznaje się za utracony.

#### *Błędy bizantyjskie (nieokreślone)*

Węzły mogą działać w absolutnie dowolny sposób, w tym próbować oszukać i zwiść inne węzły, co opisano we wcześniejszym punkcie.

W kontekście modelowania rzeczywistych systemów zwykle najbardziej przydatny jest model częściowo synchroniczny z błędami typu awaria — przywracanie stanu. Jak algorytmy rozproszone radzą sobie z pracą w tym modelu?

### **Poprawność algorytmu**

Aby zdefiniować, na czym polega *poprawność* algorytmu, można opisać jego *cechy*. Na przykład dane wyjściowe algorytmu sortującego mają tę cechę, że dla dwóch różnych pozycji z listy wyjściowej element znajdujący się bardziej po lewej stronie jest mniejszy od elementu znajdującego się bardziej po prawej. Jest to formalny sposób zdefiniowania posortowanej listy.

Można też zapisać cechy oczekiwane od algorytmu rozproszonego, aby zdefiniować jego poprawne działanie. Na przykład jeśli generowane są tokeny odgradzające dla blokad (zob. punkt „Tokeny odgradzające”), można wymagać, by algorytm miał następujące cechy:

#### *Unikatowość*

Dla żadnych dwóch żądań tokenu odgradzającego nie można zwrócić tej samej wartości.

#### *Monotoniczność sekwencji*

Jeśli dla żądania  $x$  zwrócony został token  $t_x$ , a dla żądania  $y$  token  $t_y$ , oraz żądanie  $x$  zostało przetworzone przed rozpoczęciem obsługi żądania  $y$ , to  $t_x < t_y$ .

#### *Dostępność*

Węzeł, który żąda tokenu odgradzającego i nie ulega awarii, ostatecznie otrzymuje odpowiedź.

Algorytm jest poprawny w określonym modelu systemu, jeśli zawsze zachowuje ustalone cechy we wszystkich sytuacjach, jakie według założeń mogą wystąpić w danym modelu. Dlaczego jednak jest to sensowne? Jeśli wszystkie węzły ulegną awarii lub gdy opóźnienie sieciowe stanie się nieskończenie długie, żaden algorytm nie będzie mógł wykonać swojej pracy.

## Bezpieczeństwo i aktywność

Aby jednoznacznie przedstawić omawiane zagadnienie, warto wprowadzić rozróżnienie na dwa rodzaje cech: związane z *bezpieczeństwem* i *aktywnością*. W przedstawionym przykładzie *unikatowość* i *monotoniczność sekwencji* to cechy związane z bezpieczeństwem, natomiast *dostępność* to cecha dotycząca aktywności.

Co odróżnia od siebie cechy tych dwóch typów? Wskazówką jest tu to, że cechy związane z aktywnością często obejmują w definicji słowo „ostatecznie”. I tak, dobrze zgadłeś — *spójność ostateczna* to cecha związana z aktywnością [89].

Bezpieczeństwo jest często nieformalnie definiowane jako *nie dzieje się nic złego*, a aktywność — jako *ostatecznie dzieje się coś dobrego*. Najlepiej jednak nie skupiać się na tych nieformalnych definicjach, ponieważ znaczenie słów „dobre” i „złe” jest subiektywne. Rzeczywiste definicje bezpieczeństwa i aktywności są precyzyjne oraz matematyczne [90]:

- Jeśli naruszona została cecha związana z bezpieczeństwem, można wskazać konkretny punkt w czasie, kiedy to nastąpiło. Na przykład po naruszeniu unikatowości można zidentyfikować konkretną operację, kiedy zwrócony został powtarzający się token odgradzający. Po naruszeniu takiej cechy nie można odwrócić skutków tej sytuacji — szkody już zostały wyrządzone.
- Cechy związane z aktywnością działają inaczej. Mogą nie być spełnione w określonym momencie (węzeł mógł przesłać żądanie, ale jeszcze nie otrzymał odpowiedzi), ale zawsze jest nadzieja, że zmieni się to w przyszłości (dzięki nadejściu odpowiedzi).

Zaletą podziału na cechy związane z bezpieczeństwem i aktywnością jest to, że łatwiej wtedy poradzić sobie ze skomplikowanymi modelami systemów. W algorytmach rozproszonych częstym wymogiem jest, by cechy związane z bezpieczeństwem były spełnione *zawsze*, we wszystkich sytuacjach możliwych w danym modelu systemu [88]. Oznacza to, że nawet po awarii wszystkich węzłów lub całej sieci algorytm musi gwarantować, że nie zwróci błędnego wyniku (czyli że cecha związana z bezpieczeństwem będzie spełniona).

W przypadku cech związanych z aktywnością można czynić różne zastrzeżenia. Można np. stwierdzić, że odpowiedź na żądanie musi nadejść tylko wtedy, jeśli większość węzłów nie uległa awarii i dopiero po przywróceniu stanu po przestoju sieci. Definicja modelu częściowo synchronicznego wymaga, by system ostatecznie wracał do stanu synchronicznego. Oznacza to, że zakłócenia w sieci trwają tylko skończony czas, po czym są eliminowane.

## Odwzorowywanie modeli systemu na świat rzeczywisty

Cechy związane z bezpieczeństwem i aktywnością oraz modele systemów są bardzo przydatne do analizowania poprawności algorytmów rozproszonych. Jednak w trakcie implementowania algorytmu w praktyce odczuwalne są złożone aspekty rzeczywistości i staje się jasne, że model systemu jest uproszczonym abstrakcyjnym ujęciem świata.

Na przykład w algorytmach zgodnych z modelem awarii — przywracania stanu zwykle przyjmuje się, że dane z pamięci trwałej „przeżyją” awarię. Co się jednak stanie, gdy dane na dysku zostaną uszkodzone lub zostaną wykasowane z powodu błędu sprzętowego lub niewłaściwej konfiguracji [91]? Co się stanie, jeśli oprogramowanie serwera jest błędne i po restarcie nie potrafi wykryć dysków twardych, choć są one poprawnie podłączone do serwera [92]?

W algorytmach wykorzystujących kworum (zob. punkt „Odczyty i zapisy zgodne z kworum”) przyjmuje się, że węzeł zapamiętuje dane, których zapisanie potwierdził. Jeśli węzeł może zapamiętać wcześniej zapisane dane, narusza to warunek uzyskania kworum, a tym samym i poprawność algorytmu. Możliwe, że potrzebny jest nowy model systemu, w którym zakładamy, iż pamięć trwała zwykle potrafi „przeżyć” awarie, jednak czasem dane zostają utracone. Jednak analizy w takim modelu są trudne.

W teoretycznym opisie algorytmu można umieścić założenie, że niektóre rzeczy się nie dzieją. Ponadto w systemie niebizantyjskim trzeba przyjmować pewne założenia na temat tego, jakie błędy mogą wystąpić, a jakie się nie zdarzają. Jednak w rzeczywistej implementacji niezbędny może być kod obsługujący przypadek, gdy dzieje się coś, co wedle założeń jest niemożliwe — nawet jeśli ta obsługa sprowadza się do wiersza `printf("Nie chciałbym być w Twojej skórze")` i `exit(666)`, co oznacza, że to człowiek na stanowisku operatora musi rozwiązać problem [93]. Można uznać, że na tym polega różnica między naukami komputerowymi a inżynierią oprogramowania.

Nie chcę przez to powiedzieć, że teoretyczne, abstrakcyjne modele systemów są bezwartościowe. Jest wprost przeciwnie. Takie modele są niezwykle przydatne, jeśli chcesz uprościć złożone rzeczywiste systemy do rozsądnego zestawu błędów, które można poddać analizie. Pozwala to zrozumieć problem i spróbować systematycznie go rozwiązywać. Można udowodnić poprawność algorytmów, wykazując, że ich cechy zawsze są spełnione w danym modelu systemu.

Dowiedzenie poprawności algorytmu nie oznacza, że jego *implementacja* w rzeczywistym systemie zawsze będzie działać prawidłowo. Jest to jednak bardzo dobry pierwszy krok, ponieważ analizy teoretyczne pozwalają ujawnić w algorytmie problemy, które mogłyby pozostać ukryte w systemie przez długi czas i wyrządzić szkody dopiero po złamaniu założeń (dotyczących np. synchronizacji) z powodu niezwykłych okoliczności. Analizy teoretyczne i testy empiryczne są więc równie ważne.

## Podsumowanie

W tym rozdziale omówiono wiele problemów, które mogą występować w systemach rozproszonych. Oto niektóre z tych trudności:

- Za każdym razem, gdy próbujesz przesłać pakiet w sieci, może on zostać utracony lub opóźniony o arbitralnie długi czas. Także odpowiedź może zostać utracona lub opóźniona. Dlatego jeśli nie otrzymałeś odpowiedzi, nie wiesz, czy komunikat dotarł do celu.
- Zegar węzła może być wyraźnie zdesynchronizowany względem innych węzłów (mimo dużej pracy włożonej w konfigurację NTP) oraz może nagle przeskoczyć do przodu lub wstecz. Ponadto poleganie na zegarze jest niebezpieczne, ponieważ zapewne nie masz dobrych pomiarów błędów jego wskazań.
- Proces może zostać wstrzymany na długi czas w dowolnym punkcie jego wykonywania (zapewne z powodu odzyskiwania pamięci w trybie „zatrzymać świat”), zostać uznany za niesprawny przez inne węzły, a następnie wznowić pracę bez wykrycia tego, że był wstrzymany.

To, że możliwe są tego rodzaju *awarie częściowe*, jest cechą definicyjną systemów rozproszonych. Za każdym razem, gdy oprogramowanie próbuje wykonywać operacje z udziałem innych węzłów, może zawieść, w losowym momencie zacząć pracować powoli lub w ogóle przestać reagować (i ostatecznie przekroczyć limit czasu). Twórcy systemów rozproszonych starają się wbudować odporność na awarie częściowe w oprogramowanie, tak by system jako całość mógł kontynuować pracę nawet po uszkodzeniu jego elementów.

Aby móc tolerować błędy, pierwszym krokiem jest ich *wykrywanie*. Jednak nawet to jest trudne. Większość systemów nie obejmuje precyzyjnych mechanizmów wykrywania, czy węzeł uległ awarii. Dlatego większość algorytmów rozproszonych wykorzystuje limity czasu do określania, czy zdalny węzeł wciąż jest dostępny. Ale limity czasu nie pozwalają odróżniać awarii sieci od problemów z węzłem. Ponadto zmienne opóźnienie sieciowe czasem powoduje, że węzeł jest niesłusznie uznawany za niesprawny. Zdarza się też, że węzeł działa poniżej swoich możliwości. Na przykład przepustowość gigabitowej karty sieciowej może z powodu błędu sterownika spaść nagle do 1 Kb/s [94]. Taki węzeł, który „kuleje”, ale nie jest „martwy”, może być jeszcze trudniejszy do obsłużenia niż jednoznacznie niesprawny węzeł.

Zapewnienie odporności systemu na błąd po jego wykryciu też nie jest łatwe. Nie istnieją zmienna globalna, pamięć współużytkowana, wspólne informacje ani innego rodzaju stan współużytkowany między maszynami. Węzły nie potrafią uzgodnić nawet czasu, nie mówiąc już o bardziej złożonych aspektach. Jedyny sposób przepływu informacji między węzłami polega na przesyłaniu ich w zawodnej sieci. Ważnych decyzji nie można bezpiecznie podejmować w jednym węźle, dlatego potrzebne są protokoły uzyskiwania pomocy od innych węzłów i próby osiągnięcia zgody przez kworum.

Jeśli jesteś przyzwyczajony do pisania oprogramowania w wyidealizowanym matematycznie świecie jednego komputera, gdzie te same operacje zawsze deterministycznie dają ten sam wynik, przejście do zagmatwanej fizycznej rzeczywistości systemów rozproszonych może być dla Ciebie szokiem. Z kolei inżynierowie systemów rozproszonych często uznają problem za banalny, jeśli można go rozwiązać z użyciem jednego komputera [5]. Pojedyncze komputery dają obecnie duże możliwości [95]. Jeśli możesz uniknąć otwierania puszkę Pandory i ograniczyć się do jednej maszyny, zwykle warto to zrobić.

Jednak, co opisano we wprowadzeniu do części II, skalowalność nie jest jedynym powodem, dla którego używane są systemy rozproszone. Odporność na błędy i niskie opóźnienie (dzięki umieszczeniu danych geograficznie blisko użytkowników) to równie istotne cele, a nie da się ich osiągnąć za pomocą jednego węzła.

W tym rozdziale zbadano też, czy zawodność sieci, zegarów i procesów jest niezmiennym prawem natury. Okazało się, że tak nie jest. Można zapewnić w sieciach sztywne gwarancje zwracania odpowiedzi w czasie rzeczywistym i ograniczonych opóźnień, jednak jest to bardzo kosztowne i skutkuje niższym wykorzystaniem zasobów sprzętowych. W większości systemów, w których bezpieczeństwo nie odgrywa kluczowej roli, wybierane są tańsze i zawodne rozwiązania zamiast drogich i niezawodnych.

Wspomniano tu też o superkomputerach, w których przyjmuje się, że komponenty są niezawodne, dlatego usterka jednego z nich wymaga zatrzymania i restartu całego systemu. Z kolei systemy rozproszone mogą działać w nieskończoność bez zakłóceń na poziomie usługi, ponieważ wszystkie

błędy i konserwację można obsługiwać na poziomie węzłów — przynajmniej w teorii. W praktyce jeśli we wszystkich węzłach wprowadzona zostanie błędna zmiana w konfiguracji, także system rozproszony przestanie działać.

Cały ten rozdział był poświęcony błędom i przedstawiał pesymistyczny obraz. W następnym rozdziale przechodzimy do rozwiązań. Omówiono w nim algorytmy zaprojektowane do radzenia sobie z problemami z systemów rozproszonych.

## **Literatura cytowana**

- [1] Mark Cavage, *There's Just No Getting Around It: You're Building a Distributed System*, „ACM Queue”, rocznik 11, nr 4, s. 80 – 89, kwiecień 2013 (<http://queue.acm.org/detail.cfm?id=2482856>).
- [2] Jay Kreps, *Getting Real About Distributed System Reliability*, *blog.empathybox.com*, 19 marca 2012 (<http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability>).
- [3] Sydney Padua, *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*, Particular Books, kwiecień 2015, ISBN: 978-0-141-98151-2.
- [4] Coda Hale, *You Can't Sacrifice Partition Tolerance*, *codahale.com*, 7 października 2010 (<https://codahale.com/you-cant-sacrifice-partition-tolerance/>).
- [5] Jeff Hodges, *Notes on Distributed Systems for Young Bloods*, *somethingsimilar.com*, 14 stycznia 2013 (<https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/>).
- [6] Antonio Regalado, *Who Coined „Cloud Computing”?*, *technologyreview.com*, 31 października 2011 (<https://www.technologyreview.com/s/425970/who-coined-cloud-computing/>).
- [7] Luiz André Barroso, Jimmy Clidaras i Urs Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, „Synthesis Lectures on Computer Architecture”, rocznik 8, nr 3, Morgan & Claypool Publishers, lipiec 2013, ISBN: 978-1-627-05010-4 (<http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024>).
- [8] David Fiala, Frank Mueller, Christian Engelmann i in., *Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing*, w: „International Conference for High Performance Computing, Networking, Storage and Analysis” (SC12), listopad 2012 (<http://moss.csc.ncsu.edu/~mueller/ftp/pub/mueller/papers/sc12.pdf>).
- [9] Arjun Singh, Joon Ong, Amit Agarwal i in., *Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network*, w: „Annual Conference of the ACM Special Interest Group on Data Communication” (SIGCOMM), sierpień 2015 (<http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p183.pdf>; <http://dl.acm.org/citation.cfm?doid=2785956.2787508>).
- [10] Glenn K. Lockwood, *Hadoop's Uncomfortable Fit in HPC*, *glennklockwood.blogspot.co.uk*, 16 maja 2014 (<http://glennklockwood.blogspot.co.uk/2014/05/hadoops-uncomfortable-fit-in-hpc.html>).

- [11] John von Neumann, *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*, „Automata Studies (AM-34)”, red. Claude E. Shannon i John McCarthy, Princeton University Press, 1956, ISBN:978-0-691-07916-5 ([https://ece.uwaterloo.ca/~ssundara/courses/prob\\_logics.pdf](https://ece.uwaterloo.ca/~ssundara/courses/prob_logics.pdf)).
- [12] Richard W. Hamming, *The Art of Doing Science and Engineering*, Taylor & Francis, 1997, ISBN: 978-9-056-99500-3.
- [13] Claude E. Shannon, *A Mathematical Theory of Communication*, „The Bell System Technical Journal”, rocznik 27, nr 3, s. 379 – 423 i 623 – 656, lipiec 1948 (<https://cs.brynmawr.edu/Courses/cs380/fall2012/shannon1948.pdf>).
- [14] Peter Bailis i Kyle Kingsbury, *The Network Is Reliable*, „ACM Queue”, rocznik 12, nr 7, s. 48 – 55, lipiec 2014 (<https://queue.acm.org/detail.cfm?id=2655736>).
- [15] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera i Michael Walfish, *Taming Uncertainty in Distributed Systems with Help from the Network*, w: „10th European Conference on Computer Systems” (EuroSys), kwiecień 2015 (<https://cs.nyu.edu/~mwalfish/papers/albatross-eurosys15.pdf>; <https://dl.acm.org/citation.cfm?doid=2741948.2741976>).
- [16] Phillipa Gill, Navendu Jain i Nachiappan Nagappan, *Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications*, w: „ACM SIGCOMM Conference”, sierpień 2011 (<http://conferences.sigcomm.org/sigcomm/2011/papers/sigcomm/p350.pdf>; <https://dl.acm.org/citation.cfm?doid=2018436.2018477>).
- [17] Mark Imbriaco, *Downtime Last Saturday*, [github.com](https://github.com), 26 grudnia 2012 (<https://github.com/blog/1364-downtime-last-saturday>).
- [18] Will Oremus, *The Global Internet Is Being Attacked by Sharks, Google Confirms*, [slate.com](http://www.slate.com), 15 sierpnia 2014 ([http://www.slate.com/blogs/future\\_tense/2014/08/15/shark\\_attacks\\_threaten\\_google\\_s\\_undersea\\_internet\\_cables\\_video.html](http://www.slate.com/blogs/future_tense/2014/08/15/shark_attacks_threaten_google_s_undersea_internet_cables_video.html)).
- [19] Marc A. Donges, *Re: bnx2 cards Intermittantly Going Offline*, wiadomość na liście dyskusyjnej *netdev* związanej z Linuksem, [spinics.net](http://www.spinics.net/lists/netdev/msg210485.html), 13 września 2012 (<http://www.spinics.net/lists/netdev/msg210485.html>).
- [20] Kyle Kingsbury, *Call Me Maybe: Elasticsearch*, [aphyr.com](http://aphyr.com), 15 czerwca 2014 (<https://aphyr.com/posts/317-call-me-maybe-elasticsearch>).
- [21] Salvatore Sanfilippo, *A Few Arguments About Redis Sentinel Properties and Fail Scenarios*, [antirez.com](http://antirez.com), 21 października 2014 (<http://antirez.com/news/80>).
- [22] Bert Hubert, *The Ultimate SO\_LINGER Page, or: Why Is My TCP Not Reliable*, [blog.netherlabs.nl](http://blog.netherlabs.nl), 18 stycznia 2009 ([https://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so\\_linger-page-or-why-is-my-tcp-not-reliable](https://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so_linger-page-or-why-is-my-tcp-not-reliable)).
- [23] Nicolas Liochon, *CAP: If All You Have Is a Timeout, Everything Looks Like a Partition*, [blog.thislongrun.com](http://blog.thislongrun.com), 25 maja 2015 (<http://blog.thislongrun.com/2015/05/CAP-theorem-partition-timeout-zookeeper.html>).

- [24] Jerome H. Saltzer, David P. Reed i David D. Clark, *End-To-End Arguments in System Design*, „ACM Transactions on Computer Systems”, rocznik 2, nr 4, s. 277 – 288, listopad 1984 (<http://www.ece.drexel.edu/courses/ECE-C631-501/SalRee1984.pdf>; <https://dl.acm.org/citation.cfm?doid=357401.357402>).
- [25] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog i in., *Queues Don't Matter When You Can JUMP Them!*, w: „12th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), maj 2015 ([https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-grosvenor\\_update.pdf](https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-grosvenor_update.pdf)).
- [26] Guohui Wang i T.S. Eugene Ng, *The Impact of Virtualization on Network Performance of Amazon EC2 Data Center*, w: „29th IEEE International Conference on Computer Communications” (INFOCOM), marzec 2010 (<https://www.cs.rice.edu/~eugeneng/papers/INFOCOM10-ec2.pdf>; <http://ieeexplore.ieee.org/document/5461931/>).
- [27] Van Jacobson, *Congestion Avoidance and Control*, w: „ACM Symposium on Communications Architectures and Protocols” (SIGCOMM), sierpień 1988 (<https://www.cs.usask.ca/ftp/pub/discus/seminars2002-2003/p314-jacobson.pdf>; <https://dl.acm.org/citation.cfm?doid=52324.52356>).
- [28] Brandon Philips, *etcd: Distributed Locking and Service Discovery*, w: „Strange Loop”, wrzesień 2014 (<https://www.youtube.com/watch?v=HJIjTTHWYnE>).
- [29] Steve Newman, *A Systematic Look at EC2 I/O*, *blog.scalyr.com*, 16 października 2012 (<http://blog.scalyr.com/2012/10/a-systematic-look-at-ec2-io/>).
- [30] Naohiro Hayashibara, Xavier Défago, Rami Yared i Takuya Katayama, *The  $\phi$  Accrual Failure Detector*, Japan Advanced Institute of Science and Technology, School of Information Science, raport techniczny IS-RR-2004-010, maj 2004 (<https://dspace.jaist.ac.jp/dspace/handle/10119/4784>).
- [31] Jeffrey Wang, *Phi Accrual Failure Detector*, *ternarysearch.blogspot.co.uk*, 11 sierpnia 2013 (<http://ternarysearch.blogspot.co.uk/2013/08/phi-accrual-failure-detector.html>).
- [32] Srinivasan Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*, Addison-Wesley Professional, maj 1997, ISBN: 978-0-201-63442-6.
- [33] Cisco, *Integrated Services Digital Network*, *docwiki.cisco.com* ([http://docwiki.cisco.com/wiki/Integrated\\_Services\\_Digital\\_Network](http://docwiki.cisco.com/wiki/Integrated_Services_Digital_Network)).
- [34] Othmar Kyas, *ATM Networks*, International Thomson Publishing, 1995, ISBN: 978-1-850-32128-6.
- [35] *InfiniBand FAQ*, Mellanox Technologies, 22 grudnia 2014 ([http://www.mellanox.com/related-docs/whitepapers/InfiniBandFAQ\\_FQ\\_100.pdf](http://www.mellanox.com/related-docs/whitepapers/InfiniBandFAQ_FQ_100.pdf)).
- [36] Jose Renato Santos, Yoshio Turner i G. (John) Janakiraman, *End-to-End Congestion Control for InfiniBand*, w: „22nd Annual Joint Conference of the IEEE Computer and Communications Societies” (INFOCOM), kwiecień 2003. Opublikowane też przez HP Laboratories Palo Alto, raport techniczny HPL-2002-359 (<http://www.hpl.hp.com/techreports/2002/HPL-2002-359.pdf>; <http://ieeexplore.ieee.org/document/1208949/>).



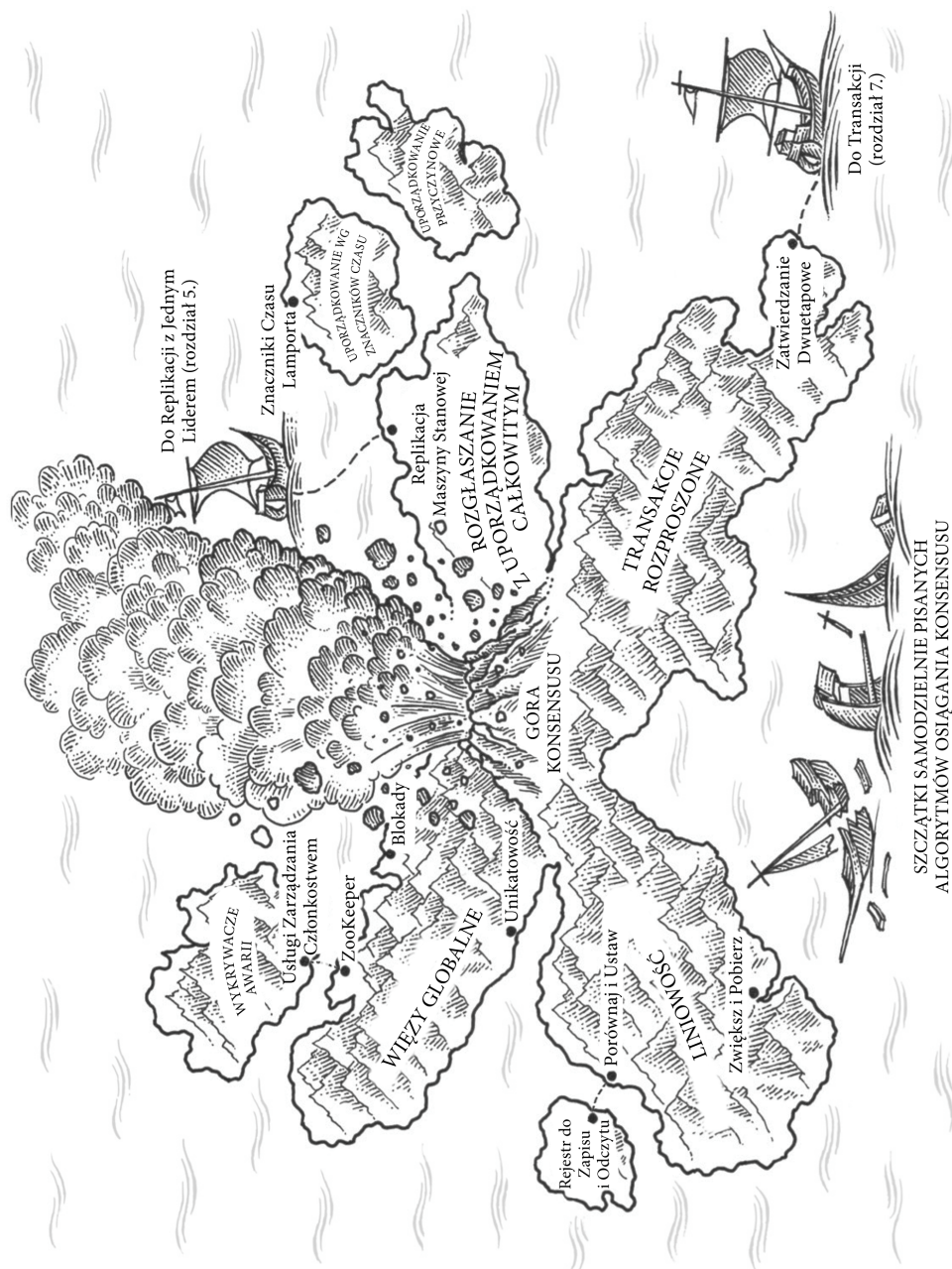
- [37] Ulrich Windl, David Dalton, Marc Martinec i Dale R. Worley, *The NTP FAQ and HOWTO*, ntp.org, listopad 2006 (<http://www.ntp.org/ntpfaq/NTP-a-faq.htm>).
- [38] John Graham-Cumming, *How and why the leap second affected Cloudflare DNS*, blog.cloudflare.com, 1 stycznia 2017 (<https://blog.cloudflare.com/how-and-why-the-leap-second-affected-cloudflare-dns/>).
- [39] David Holmes, *Inside the Hotspot VM: Clocks, Timers and Scheduling Events — Part I — Window*, blogs.oracle.com, 2 października 2006 ([https://blogs.oracle.com/dholmes/entry/inside\\_the\\_hotspot\\_vm\\_clocks](https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks)).
- [40] Steve Loughran, *Time on Multi-Core, Multi-Socket Servers*, steveloughran.blogspot.co.uk, 17 września 2015 (<http://steveloughran.blogspot.co.uk/2015/09/time-on-multi-core-multi-socket-servers.html>).
- [41] James C. Corbett, Jeffrey Dean, Michael Epstein i in., *Spanner: Google's Globally-Distributed Database*, w: „10th USENIX Symposium on Operating System Design and Implementation” (OSDI), październik 2012 (<https://research.google.com/archive/spanner.html>).
- [42] M. Caporali i R. Ambrosini, *How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet?*, „European Journal of Physics”, rocznik 23, nr 4, s. L17 – L21, czerwiec 2012 (<http://iopscience.iop.org/article/10.1088/0143-0807/23/4/103/meta>).
- [43] Nelson Minar, *A Survey of the NTP Network*, alumni.media.mit.edu, grudzień 1999 (<http://alumni.media.mit.edu/~nelson/research/ntp-survey99/>).
- [44] Viliam Holub, *Synchronizing Clocks in a Cassandra Cluster Pt. 1 — The Problem*, blog.logentries.com, 14 marca 2014 (<https://blog.rapid7.com/2014/03/14/synchronizing-clocks-in-a-cassandra-cluster-pt-1-the-problem/>).
- [45] Poul-Henning Kamp, *The One-Second War (What Time Will You Die?)*, „ACM Queue”, rocznik 9, nr 4, s. 44 – 48, kwiecień 2011 (<http://queue.acm.org/detail.cfm?id=1967009>; <https://dl.acm.org/citation.cfm?doid=1966989.1967009>).
- [46] Nelson Minar, *Leap Second Crashes Half the Internet*, somebits.com, 3 lipca 2012 (<http://www.somebits.com/weblog/tech/bad/leap-second-2012.html>).
- [47] Christopher Pascoe, *Time, Technology and Leaping Seconds*, googleblog.blogspot.co.uk, 15 września 2011 (<https://googleblog.blogspot.co.uk/2011/09/time-technology-and-leaping-seconds.html>).
- [48] Mingxue Zhao i Jeff Barr, *Look Before You Leap — The Coming Leap Second and AWS*, aws.amazon.com, 18 maja 2015 (<https://aws.amazon.com/blogs/aws/look-before-you-leap-the-coming-leap-second-and-aws/>).
- [49] Darryl Veitch i Kanthaiah Vijayalayan, *Network Timing and the 2015 Leap Second*, w: „17th International Conference on Passive and Active Measurement” (PAM), kwiecień 2016 ([http://crin.eng.uts.edu.au/~darryl/Publications/LeapSecond\\_camera.pdf](http://crin.eng.uts.edu.au/~darryl/Publications/LeapSecond_camera.pdf); [https://link.springer.com/chapter/10.1007%2F978-3-319-30505-9\\_29](https://link.springer.com/chapter/10.1007%2F978-3-319-30505-9_29)).

- [50] *Timekeeping in VMware Virtual Machines*, Information Guide, VMware, Inc., grudzień 2011 (<https://www.vmware.com/techpapers/2006/timekeeping-in-vmware-virtual-machines-238.html>).
- [51] *MiFID II/MiFIR: Regulatory Technical and Implementing Standards — Annex I (Draft)*, European Securities and Markets Authority, raport ESMA/2015/1464, wrzesień 2015 ([https://www.esma.europa.eu/sites/default/files/library/2015/11/2015-esma-1464\\_annex\\_i\\_-\\_draft\\_rts\\_and\\_its\\_on\\_mifid-ii\\_and\\_mifir.pdf](https://www.esma.europa.eu/sites/default/files/library/2015/11/2015-esma-1464_annex_i_-_draft_rts_and_its_on_mifid-ii_and_mifir.pdf)).
- [52] Luke Bigum, *Solving MiFID II Clock Synchronisation With Minimum Spend (Part 1)*, *lmax.com*, 27 listopada 2015 (<https://www.lmax.com/blog/staff-blogs/2015/11/27/solving-mifid-ii-clock-synchronisation-minimum-spend-part-1/>).
- [53] Kyle Kingsbury, *Call Me Maybe: Cassandra*, *aphyr.com*, 24 września 2013 (<https://aphyr.com/posts/294-call-me-maybe-cassandra/>).
- [54] John Daily, *Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems*, *basho.com*, 12 listopada 2013 (<http://basho.com/posts/technical/clocks-are-bad-or-welcome-to-distributed-systems/>).
- [55] Kyle Kingsbury, *The Trouble with Timestamps*, *aphyr.com*, 12 października 2013 (<https://aphyr.com/posts/299-the-trouble-with-timestamps>).
- [56] Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, „Communications of the ACM”, rocznik 21, nr 7, s. 558 – 565, lipiec 1978 (<https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Fflamport%2Fpubs%2Ftime-clocks.pdf;https://dl.acm.org/citation.cfm?doid=359545.359563>).
- [57] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa i in., *Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases*, State University of New York at Buffalo, Computer Science and Engineering Technical Report 2014-04, maj 2014 (<https://www.cse.buffalo.edu/tech-reports/2014-04.pdf>).
- [58] Justin Sheehy, *There Is No Now: Problems With Simultaneity in Distributed Systems*, „ACM Queue”, rocznik 13, nr 3, s. 36 – 41, marzec 2015 (<https://queue.acm.org/detail.cfm?id=2745385;https://dl.acm.org/citation.cfm?doid=2766485.2733108>).
- [59] Murat Demirbas, *Spanner: Google’s Globally-Distributed Database*, *muratbuffalo.blogspot.co.uk*, 4 lipca 2013 ([http://muratbuffalo.blogspot.co.uk/2013/07/spanner-google-globally-distributed\\_4.html](http://muratbuffalo.blogspot.co.uk/2013/07/spanner-google-globally-distributed_4.html)).
- [60] Dahlia Malkhi i Jean-Philippe Martin, *Spanner’s Concurrency Control*, „ACM SIGACT News”, rocznik 44, nr 3, s. 73 – 77, wrzesień 2013 (<http://www.cs.cornell.edu/~ie53/publications/DC-col51-Sep13.pdf>; <https://dl.acm.org/citation.cfm?doid=2527748.2527767>).
- [61] Manuel Bravo, Nuno Diegues, Jingna Zeng i in., *On the Use of Clocks to Enforce Consistency in the Cloud*, „IEEE Data Engineering Bulletin”, rocznik 38, nr 1, s. 18 – 31, marzec 2015 (<http://sites.computer.org/debull/A15mar/p18.pdf>).

- [62] Spencer Kimball, *Living Without Atomic Clocks*, *cockroachlabs.com*, 17 lutego 2016 (<https://www.cockroachlabs.com/blog/living-without-atomic-clocks/>).
- [63] Cary G. Gray i David R. Cheriton, *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency*, w: „12th ACM Symposium on Operating Systems Principles” (SOSP), grudzień 1989 (<https://dl.acm.org/citation.cfm?doid=74850.74870>).
- [64] Todd Lipcon, *Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1*, *blog.cloudera.com*, 24 lutego 2011 (<http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/>).
- [65] Martin Thompson, *Java Garbage Collection Distilled*, *mechanicalsympathy.blogspot.co.uk*, 16 lipca 2013 (<https://mechanical-sympathy.blogspot.co.uk/2013/07/java-garbage-collection-distilled.html>).
- [66] Alexey Ragozin, *How to Tame Java GC Pauses? Surviving 16GiB Heap and Greater*, *java.dzone.com*, 28 czerwca 2011 (<https://dzone.com/articles/how-tame-java-gc-pauses>).
- [67] Christopher Clark, Keir Fraser, Steven Hand i in., *Live Migration of Virtual Machines*, w: „2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation” (NSDI), maj 2005 (<http://www.cl.cam.ac.uk/research/srg/netos/papers/2005-nsdi-migration.pdf>).
- [68] Mike Shaver, *fsyncers and Curveballs*, *shaver.off.net*, 25 maja 2008 (<http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/>).
- [69] Zhenyun Zhuang i Cuong Tran, *Eliminating Large JVM GC Pauses Caused by Background IO Traffic*, *engineering.linkedin.com*, 10 lutego 2016 (<https://engineering.linkedin.com/blog/2016/02/eliminating-large-jvm-gc-pauses-caused-by-background-io-traffic>).
- [70] David Terei i Amit Levy, *Blade: A Data Center Garbage Collector*, arXiv: 1504.02578, 13 kwietnia 2015 (<https://arxiv.org/pdf/1504.02578.pdf>).
- [71] Martin Maas, Tim Harris, Krste Asanović i John Kubiawicz, *Trash Day: Coordinating Garbage Collection in Distributed Systems*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<https://timharris.uk/papers/2015-hotos.pdf>).
- [72] *Predictable Low Latency*, Cinnober Financial Technology AB, *cinnober.com*, 24 listopada 2013 ([https://cdn2.hubspot.net/hubfs/1624455/Website\\_2016/Content/Whitepapers/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf](https://cdn2.hubspot.net/hubfs/1624455/Website_2016/Content/Whitepapers/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf)).
- [73] Martin Fowler, *The LMAX Architecture*, *martinfowler.com*, 12 lipiec 2011 (<https://martinfowler.com/articles/lmax.html>).
- [74] Flavio P. Junqueira i Benjamin Reed, *ZooKeeper: Distributed Process Coordination*, O’Reilly Media, 2013, ISBN: 978-1-449-36130-3.
- [75] Enis Söztutar, *HBase and HDFS: Understanding Filesystem Usage in HBase*, w: „HbaseCon”, czerwiec 2013 (<https://www.slideshare.net/enisoz/hbase-and-hdfs-understanding-filesystem-usage>).
- [76] Caitie McCaffrey, *Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived*, *caitiem.com*, 23 czerwca 2015 (<https://caitiem.com/2015/06/23/clients-are-jerks-aka-how-halo-4-dosed-the-services-at-launch-how-we-survived/>).

- [77] Leslie Lamport, Robert Shostak i Marshall Pease, *The Byzantine Generals Problem*, „ACM Transactions on Programming Languages and Systems” (TOPLAS), rocznik 4, nr 3, s. 382 – 401, lipiec 1982 (<https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Fflamport%2Fpubs%2Fbyz.pdf>; <https://dl.acm.org/citation.cfm?doid=357172.357176>).
- [78] Jim N. Gray, *Notes on Data Base Operating Systems*, „Operating Systems: An Advanced Course”, Lecture Notes in Computer Science, rocznik 60, red. Rudolf Bayer, Robert M. Graham i G. Seegmüller, s. 393 – 481, Springer-Verlag, 1978, ISBN: 978-3-540-08755-7 (<http://jimgray.azurewebsites.net/papers/dbos.pdf>).
- [79] Brian Palmer, *How Complicated Was the Byzantine Empire?*, *slate.com*, 20 października 2011 ([http://www.slate.com/articles/news\\_and\\_politics/explainer/2011/10/the\\_byzantine\\_tax\\_code\\_how\\_complicated\\_was\\_byzantium\\_anyway\\_.html](http://www.slate.com/articles/news_and_politics/explainer/2011/10/the_byzantine_tax_code_how_complicated_was_byzantium_anyway_.html)).
- [80] Leslie Lamport, *My Writings*, *research.microsoft.com*, 16 grudnia 2014 (<http://lamport.azurewebsites.net/pubs/pubs.html>); tę stronę można znaleźć, wpisując w wyszukiwarce 23-znakowy łańcuch uzyskany w wyniku usunięcia myślników z łańcucha `all1a-mport-spubso-ntheweb`.
- [81] John Rushby, *Bus Architectures for Safety-Critical Embedded Systems*, w: „1st International Workshop on Embedded Software” (EMSOFT), październik 2001 (<http://www.csl.sri.com/papers/emsoft01/emsoft01.pdf>).
- [82] Jake Edge, *ELC: SpaceX Lessons Learned*, *lwn.net*, 6 marca 2013 (<https://lwn.net/Articles/540368/>).
- [83] Andrew Miller i Joseph J. LaViola, Jr., *Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin*, University of Central Florida, raport techniczny CS-TR-14-01, kwiecień 2014 (<http://nakamotoinstitute.org/static/docs/anonymous-byzantine-consensus.pdf>).
- [84] James Mickens, *The Saddest Moment*, „USENIX ;login: logout”, maj 2013 ([https://www.usenix.org/system/files/login-logout\\_1305\\_mickens.pdf](https://www.usenix.org/system/files/login-logout_1305_mickens.pdf)).
- [85] Evan Gilman, *The Discovery of Apache ZooKeeper’s Poison Packet*, *pagerduty.com*, 7 maja 2015 (<https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet/>).
- [86] Jonathan Stone i Craig Partridge, *When the CRC and TCP Checksum Disagree*, w: „ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication” (SIGCOMM), sierpień 2000 (<http://conferences.sigcomm.org/sigcomm/2000/conf/paper/sigcomm2000-9-1.pdf>; <https://dl.acm.org/citation.cfm?doid=347059.347561>).
- [87] Evan Jones, *How Both TCP and Ethernet Checksums Fail*, *evanjones.ca*, 5 października 2015 (<http://www.evanjones.ca/tcp-and-ethernet-checksums-fail.html>).
- [88] Cynthia Dwork, Nancy Lynch i Larry Stockmeyer, *Consensus in the Presence of Partial Synchrony*, „Journal of the ACM”, rocznik 35, nr 2, s. 288 – 323, kwiecień 1988 (<https://www.net.t-labs.tu-berlin.de/~petr/ADC-07/papers/DLS88.pdf>; <https://dl.acm.org/citation.cfm?doid=42282.42283>).

- [89] Peter Bailis i Ali Ghodsi, *Eventual Consistency Today: Limitations, Extensions, and Beyond*, „ACM Queue”, rocznik 11, nr 3, s. 55 – 63, marzec 2013 (<http://queue.acm.org/detail.cfm?id=2462076>; <https://dl.acm.org/citation.cfm?doid=2460276.2462076>).
- [90] Bowen Alpern i Fred B. Schneider, *Defining Liveness*, „Information Processing Letters”, rocznik 21, nr 4, s. 181 – 185, październik 1985 (<https://www.cs.cornell.edu/fbs/publications/DefLiveness.pdf>; <http://www.sciencedirect.com/science/article/pii/0020019085900560?via%3Dihub>).
- [91] Flavio P. Junqueira, *Dude, Where’s My Metadata?*, *fpj.me*, 28 maja 2015 (<https://fpj.me/2015/05/28/dude-wheres-my-metadata/>).
- [92] Scott Sanders, *January 28th Incident Report*, *github.com*, 3 lutego 2016 (<https://github.com/blog/2106-january-28th-incident-report>).
- [93] Jay Kreps, *A Few Notes on Kafka and Jepsen*, *blog.empathybox.com*, 25 września 2013 (<http://blog.empathybox.com/post/62279088548/a-few-notes-on-kafka-and-jepsen>).
- [94] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa i in., *Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems*, w: „4th ACM Symposium on Cloud Computing” (SoCC), październik 2013 (<http://ucare.cs.uchicago.edu/pdf/socc13-limplock.pdf>; <https://dl.acm.org/citation.cfm?doid=2523616.2523627>).
- [95] Frank McSherry, Michael Isard i Derek G. Murray, *Scalability! But at What COST?*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<http://www.frankmcsherry.org/assets/COST.pdf>).



SZCZĄTKI SAMODZIELNIE PISANYCH  
ALGORYTMÓW OSIĄGANIA KONSENSUSU

## Spójność i konsensus

*Lepiej być żywym i się mylić czy mieć rację i być martwym?*

— Jay Kreps, *A Few Notes on Kafka and Jepsen* (2013)

W rozdziale 8. napisano, że w systemach rozproszonych może wystąpić wiele problemów. Najprostszą metodą radzenia sobie z takimi usterkami jest pozwolenie na awarię całej usługi i wyświetlanie użytkownikowi komunikatu o błędzie. Jeśli to rozwiązanie jest nieakceptowalne, trzeba znaleźć sposób na *tolerowanie* błędów, czyli zapewnienie prawidłowej pracy usługi nawet wtedy, gdy jakiś wewnętrzny komponent działa nieprawidłowo.

W tym rozdziale opisane są przykładowe algorytmy i protokoły służące do tworzenia systemów rozproszonych odpornych na błędy. Zakładamy, że występować mogą wszystkie problemy wymienione w rozdziale 8. Pakiety mogą zostać utracone, przedstawione, zduplikowane lub opóźnione o dowolny czas w sieci. Zegary w najlepszym scenariuszu zwracają przybliżony czas, a węzły w dowolnym momencie mogą zostać wstrzymane (np. z powodu odzyskiwania pamięci) lub ulec awarii.

Najlepszy sposób na budowanie systemów odpornych na błędy polega na znalezieniu abstrakcji ogólnego przeznaczenia z przydatnymi gwarancjami, zaimplementowaniu ich raz i poleganiu na tych gwarancjach w aplikacjach. To samo podejście zastosowano w kontekście transakcji w rozdziale 7. Dzięki wykorzystaniu transakcji aplikacja może „udawać”, że nie występują awarie (atomowość), że nikt inny nie korzysta jednocześnie z bazy (izolacja) i że urządzenia przechowujące dane są w pełni niezawodne (trwałość). Choć awarie, sytuacje wyścigu i błędy dysków się zdarzają, abstrakcja w postaci transakcji ukrywa te problemy, dzięki czemu aplikacja nie musi się nimi przejmować.

Teraz zastosujemy podobne podejście i poszukamy abstrakcji, które umożliwią aplikacji ignorowanie niektórych problemów w systemach rozproszonych. Na przykład jedną z najważniejszych abstrakcji z obszaru systemów rozproszonych jest *konsensus* związany z uzgadnianiem czegoś przez wszystkie węzły. W tym rozdziale zobaczysz, że niezawodne osiągnięcie konsensusu mimo błędów sieci i awarii procesów to zaskakująco trudne zadanie.

Po zaimplementowaniu osiągnięcia konsensusu aplikacje mogą korzystać z tego mechanizmu w różnych celach. Załóżmy, że używasz bazy z replikacją z jednym liderem. Jeśli lider przestanie działać i konieczne będzie przełączenie awaryjne, pozostałe węzły bazy mogą wybrać nowego lidera na podstawie konsensusu. W punkcie „Radzenie sobie z przestojami węzłów” wyjaśniono, że ważne

jest, aby działał tylko jeden lider i by wszystkie węzły się zgadzały, kto nim jest. Jeśli dwa węzły uważają, że są liderem, następuje *rozszczerzenie mózgu*, co często prowadzi do utraty danych. Poprawna implementacja osiągnięcia konsensusu pomaga uniknąć takich problemów.

Dalej w rozdziale, w punkcie „Transakcje rozproszone a konsensus”, przyjrzymy się algorytmom rozwiązującym problem osiągnięcia konsensusu i powiązane z tym trudności. Najpierw jednak trzeba zapoznać się z zestawem gwarancji i abstrakcji z systemów rozproszonych.

Trzeba zrozumieć zakres tego, co jest możliwe, a co nie. W niektórych sytuacjach system może tolerować błędy i kontynuować pracę. W innych scenariuszach jest to niemożliwe. Granice tego, co jest możliwe, zostały szczegółowo zbadane zarówno za pomocą dowodów teoretycznych, jak i w praktycznych implementacjach. Przegląd tych podstawowych ograniczeń znajdziesz dalej w rozdziale.

Badacze z dziedziny systemów rozproszonych zajmują się omawianymi tu kwestiami od dziesięcioleci, dlatego ilość dostępnego materiału jest duża. Tu możemy przedstawić tylko niewielką część informacji. W tej książce nie ma miejsca na szczegółowe omawianie formalnych modeli i dowodów, dlatego ograniczymy się do nieformalnych intuicji. Jeśli ten temat Cię interesuje, w literaturze cytowanej znajdziesz dużo dodatkowej wiedzy.

## Gwarancje spójności

W punkcie „Problemy z opóźnieniem replikacji” opisano problemy z synchronizacją występujące w bazach z replikacją. Jeśli w tym samym momencie przyjrysz się dwóm węzłom bazy, zapewne zobaczysz w nich różne dane. Jest tak, ponieważ żądania zapisu docierają do różnych węzłów w innym czasie. Te niespójności pojawiają się niezależnie od używanej w bazie metody replikacji (z jednym liderem, z wieloma liderami lub bez lidera).

Większość baz z replikacją zapewnia przynajmniej *spójność ostateczną*. To oznacza, że jeśli przestaniesz zapisywać dane w bazie i odczekasz nieokreśloną ilość czasu, ostatecznie wszystkie żądania odczytu będą zwracać tę samą wartość [1]. Oznacza to, że niespójność jest tymczasowa i ostatecznie zostaje wyeliminowana (przy założeniu, że błędy w sieci też zostaną ostatecznie naprawione). Lepszą nazwą dla spójności ostatecznej może być *konwergencja*, ponieważ oczekuje się, że wszystkie repliki ostatecznie dojdą do tej samej wartości [2].

Jest to jednak bardzo słaba gwarancja, która nic nie mówi na temat tego, *kiedy* repliki staną się spójne. Do czasu uzyskania spójności odczyty mogą zwracać cokolwiek (lub nawet nic nie zwracać) [1]. Na przykład jeśli zapiszesz wartość, po czym natychmiast ją wczytasz, nie ma gwarancji, że zobaczysz zapisane dane, ponieważ odczyt może być obsługiwany przez inną replikę (zob. punkt „Odczyt własnych zapisów”).

Spójność ostateczna jest niewygodna dla programistów aplikacji, ponieważ znacznie się różni od działania zmiennych w normalnych programach jednowątkowych. Jeśli przypiszesz zmiennej wartość, a następnie szybko ją wczytasz, nie oczekujesz, że zwrócona zostanie stara wartość lub że odczyt zakończy się niepowodzeniem. Baza na pozór przypomina zmienną, którą możesz wczytywać i zapisywać, jednak w rzeczywistości działa w dużo bardziej skomplikowany sposób [3].



Gdy używasz bazy, która zapewnia tylko słabe gwarancje, musisz być stale świadom jej ograniczeń i nie przyjmować przypadkowo zbyt wygórowanych założeń. Błędy są często subtelne i trudne do wykrycia w czasie testów, ponieważ przez większość czasu aplikacja może działać poprawnie. Przypadki brzegowe związane ze spójnością ostateczną uwidaczniają się tylko po wystąpieniu błędu w systemie (np. po zakłóceniu pracy sieci) lub przy wysokiej współbieżności.

W tym rozdziale analizowane są mocniejsze modele spójności niż te, jakie mogą być dostępne w systemach danych. Te mocniejsze modele wiążą się jednak z kosztami. Systemy z silniejszymi gwarancjami bywają mniej wydajne i mniej odporne na błędy niż systemy zapewniające słabsze gwarancje. Mimo to silniejsze gwarancje mogą być atrakcyjne, ponieważ ułatwiają poprawne korzystanie z systemu. Gdy zapoznasz się z kilkoma różnymi modelami spójności, łatwiej Ci będzie zdecydować, który z nich najlepiej pasuje do Twoich potrzeb.

Występują pewne podobieństwa między modelami spójności w środowisku rozproszonym a opisaną wcześniej hierarchią poziomów izolacji transakcji [4, 5] (zob. punkt „Niskie poziomy izolacji”). Jednak choć te zagadnienia częściowo się pokrywają, są w dużym stopniu niezależne od siebie. Izolacja transakcji ma przede wszystkim służyć unikaniu sytuacji wyścigu spowodowanej jednoczesnym wykonywaniem transakcji, natomiast spójność w środowisku rozproszonym ma głównie zapewniać koordynację stanu replik w obliczu opóźnień i błędów.

W tym rozdziale opisano wiele zagadnień, jednak — jak zobaczysz — są one ściśle powiązane ze sobą:

- Najpierw opisany jest najmocniejszy z powszechnie stosowanych modeli spójności, model *liniowy*, wraz z jego wadami i zaletami.
- Dalej omówiona jest kwestia porządkowania zdarzeń w systemie rozproszonym („Gwarancje kolejności”), głównie w związku z przyczynowością i uporządkowaniem całkowitym.
- W trzeciej sekcji („Transakcje rozproszone a konsensus”) objaśniono, jak atomowo zatwierdzać transakcję rozproszoną, co ostatecznie doprowadzi do rozwiązań problemu osiągnięcia konsensusu.

## Liniowość

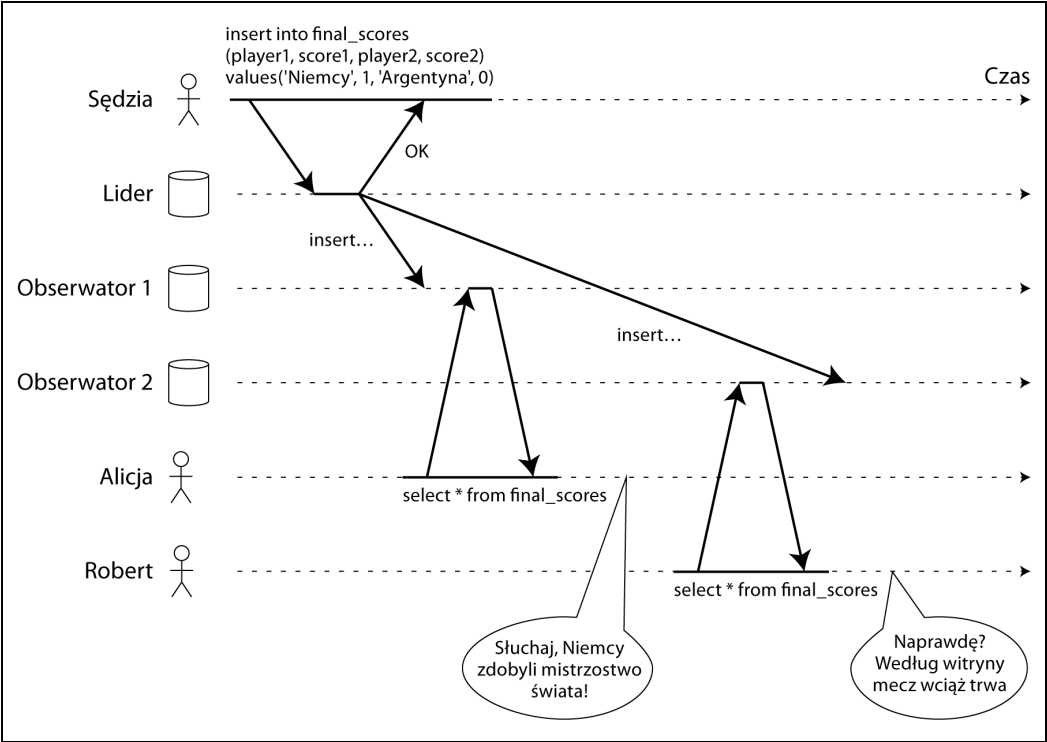
Jeśli w ostatecznie spójnej bazie zadasz dwóm replikom jednocześnie to samo pytanie, możesz uzyskać dwie różne odpowiedzi. Jest to dezorientujące. Czy nie byłoby znacznie prościej, gdyby baza mogła dawać iluzję tego, że istnieje tylko jedna replika (czyli jedna kopia danych)? Wtedy każdy klient miałby ten sam obraz danych i nie musiałbyś się martwić o opóźnienie replikacji.

Na tym pomysł oparta jest *liniowość* [6] (inne określenia to: *spójność atomowa* [7], *spójność silna*, *spójność natychmiastowa* lub *spójność zewnętrzna* [8]). Pełna definicja liniowości jest skomplikowana. Przeanalizowano ją w dalszej części punktu. Podstawowy pomysł polega na tym, aby system wyglądał tak, jakby istniała tylko jedna kopia danych, a wszystkie wykonywane na niej operacje były atomowe. Jeśli zapewniona jest taka gwarancja, to choć w rzeczywistości może istnieć wiele replik, aplikacja nie musi się tym przejmować.

W systemie liniowym bezpośrednio po tym, jak jeden klient z powodzeniem przeprowadzi zapis, wszystkie klienty wczytujące dane z bazy muszą móc zobaczyć właśnie zapisaną wartość. Utrzymywanie iluzji istnienia jednej kopii danych wymaga gwarantowania, że wczytywana jest najnowsza,

aktualna wartość, która nie pochodzi z nieaktualnej pamięci podręcznej lub repliki. Oznacza to, że liniowość to *gwarancja nowości*. Aby dobrze zrozumieć tę kwestię, warto się przyjrzeć systemowi, który nie jest liniowy.

Na rysunku 9.1 pokazana jest przykładowa nieliniowa witryna dla kibiców [9]. Alicja i Robert znajdują się w tym samym pomieszczeniu i sprawdzają w telefonach wynik finału Mistrzostw Świata w Piłce Nożnej z 2014 r. Bezpośrednio po podaniu wyniku końcowego Alicja odświeża stronę, widzi, że podano zwycięzcę, i podekscytowana informuje o tym Roberta. Robert z niedowierzaniem odświeża stronę w swoim telefonie, jednak jego żądanie trafia do opóźnionej repliki bazy. Dla tego witryna informuje, że mecz wciąż trwa.



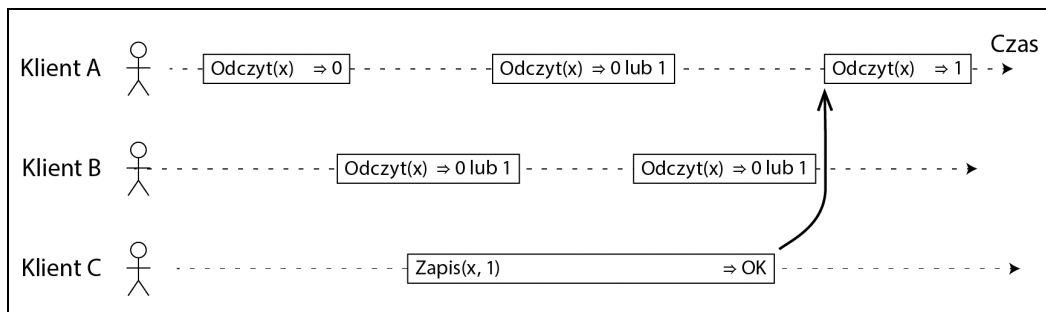
Rysunek 9.1. Ten system nie jest liniowy, przez co kibice mogą być zdezorientowani

Gdyby Alicja i Robert odświeżyli stronę w tym samym momencie, uzyskanie dwóch różnych wyników byłoby mniejszym zaskoczeniem, ponieważ nie wiedzieliby, kiedy dokładnie ich żądania zostały przetworzone przez serwer. Jednak Robert wie, że wcisnął przycisk odświeżania (czyli zainicjował zapytanie) po tym, jak usłyszał Alicję wykrzykującą końcowy wynik. Dlatego Robert oczekuje, że wynik dla jego zapytania będzie przynajmniej równie aktualny jak odpowiedź uzyskana przez Alicję. Zwrócenie nieaktualnego wyniku dla zapytania Roberta jest naruszeniem liniowości.

## Co sprawia, że system jest liniowy?

Podstawowa reguła stojąca za liniowością jest prosta — system powinien działać tak, jakby istniała tylko jedna kopia danych. Jednak precyzyjne określenie tego, co to oznacza, wymaga staranności. Aby lepiej zrozumieć liniowość, przyjrzyj się kilku przykładom.

Na rysunku 9.2 przedstawione są trzy klienty jednocześnie wczytujące i zapisujące ten sam klucz  $x$  w liniowej bazie danych. W literaturze z obszaru systemów rozproszonych  $x$  jest nazywany *rejestrem*. W praktyce może to być np. jeden klucz w bazie z kluczami i wartościami, jeden wiersz w bazie relacyjnej lub jeden dokument w bazie dokumentowej.



Rysunek 9.2. Jeśli żądanie odczytu jest zgłaszane jednocześnie z żądaniem zapisu, może zwrócić starą lub nową wartość

Na rysunku 9.2 dla uproszczenia żądania są pokazane tylko z punktu widzenia klientów, a nie z perspektywy wewnętrznych mechanizmów bazy. Każdy prostokąt reprezentuje żądanie zgłoszone przez klienta. Początek prostokąta to czas przesłania żądania, a koniec to moment otrzymania odpowiedzi przez klienta. Z powodu zmiennych opóźnień w sieci klient nie wie, kiedy baza przetworzyła żądanie. Wie tylko, że musiało się to stać pomiędzy wysłaniem żądania a otrzymaniem odpowiedzi<sup>1</sup>.

W tym przykładzie na rejestrze wykonywane są dwa typy operacji:

- $\text{odczyt}(x) \Rightarrow v$  oznacza, że klient zażądał odczytu wartości rejestru  $x$ , a baza zwróciła wartość  $v$ .
- $\text{zapis}(x, v) \Rightarrow r$  oznacza, że klient zażądał ustawienia rejestru  $x$  na wartość  $v$ , a baza zwróciła odpowiedź  $r$  (może nią być OK lub błąd).

Na rysunku 9.2 wartość rejestru  $x$  wynosi początkowo 0, a klient C wykonuje żądanie zapisu, aby ustawić ją na 1. W trakcie wykonywania tej operacji klienty A i B kilkakrotnie kierują zapytanie do bazy, aby wczytać najnowszą wartość. Jakie odpowiedzi A i B mogą uzyskać na żądania odczytu?

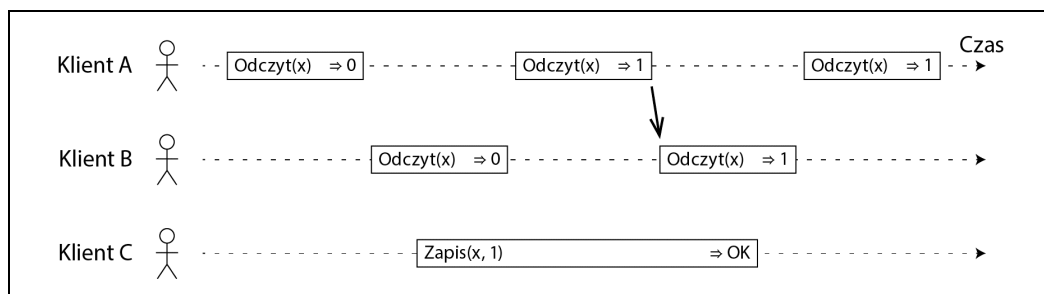
- Pierwsza operacja odczytu wykonywana przez A kończy się przed rozpoczęciem zapisu, dlatego z pewnością zwrócona zostanie dawna wartość, 0.

<sup>1</sup> Subtelnym szczegółem na rysunku jest przyjęcie istnienia globalnego zegara reprezentowanego przez oś poziomą. Choć w rzeczywistych systemach zwykle nie ma precyzyjnych zegarów (zob. punkt „Zawodne zegary”), takie założenie jest dopuszczalne. Na potrzeby analizy algorytmu rozproszonego można udać, że istnieje precyzyjny zegar globalny, przy czym algorytm nie może mieć do niego dostępu [47]. Algorytm może widzieć tylko przybliżenie rzeczywistego czasu generowane przez oscylator kwarcowy i serwery NTP.

- Ostatni odczyt wykonywany przez A zaczyna się po zakończeniu zapisu, dlatego jeśli baza jest liniowa, z pewnością trzeba zwrócić nową wartość, 1. Wiadomo, że zapis musiał zostać przetworzony między rozpoczęciem a zakończeniem operacji zapisu, a odczyt — między rozpoczęciem a zakończeniem operacji odczytu. Skoro odczyt rozpoczął się po zakończeniu zapisu, odczyt musiał zostać przetworzony po zapisie. Dlatego widoczna musi być nowo zapisana wartość.
- Wszystkie operacje odczytu pokrywające się w czasie z operacją zapisu mogą zwrócić 0 lub 1, ponieważ nie wiadomo, czy zapis został już wprowadzony w momencie przetwarzania operacji odczytu. Takie operacje są *jednoczesne* względem zapisu.

To jednak nie wystarczy do kompletnego opisu liniowości. Gdyby odczyty jednoczesne względem zapisu mogły zwracać albo starą, albo nową wartość, jednostki wczytujące mogłyby w trakcie zapisu widzieć kilkakrotne przeskoki między dawnymi a nowymi danymi. Nie tego oczekujemy od systemu, który symuluje „jedną kopię danych”<sup>2</sup>.

Aby zapewnić liniowość tego systemu, trzeba dodać następne ograniczenie, przedstawione na rysunku 9.3.



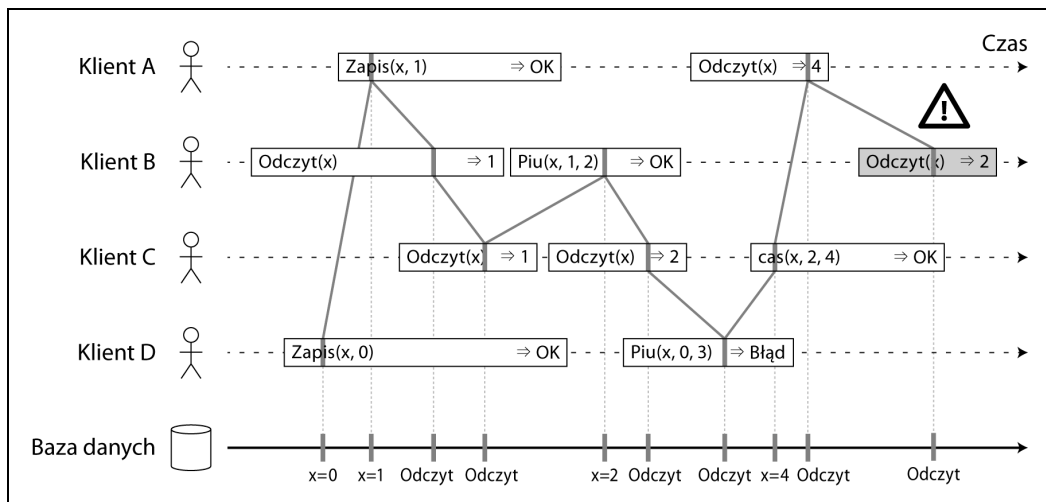
Rysunek 9.3. Po tym, jak jeden odczyt zwrócił nową wartość, wszystkie późniejsze odczyty (zarówno w tym samym kliencie, jak i w innych) też muszą ją zwracać

W systemach liniowych musi istnieć punkt w czasie (między rozpoczęciem a zakończeniem operacji zapisu), kiedy wartość  $x$  atomowo zmienia się z 0 w 1. Dlatego jeśli odczyt jednego klienta zwraca nową wartość 1, wszystkie późniejsze odczyty też muszą zwracać tę nową wartość — nawet jeśli operacja zapisu nie została jeszcze ukończona.

Zależności czasowe są na rysunku 9.3 pokazane za pomocą strzałki. Klient A jako pierwszy wczytuje nową wartość (1). Zaraz po tym, jak dla odczytu klienta A zostaje zwrócona wartość, klient B rozpoczyna nowy odczyt. Ponieważ odczyt klienta B odbywa się po odczycie klienta A, także musi dać wartość 1, choć zapis wykonywany przez klienta C wciąż jest w toku. Jest to ta sama sytuacja co z Alicją i Robertem z rysunku 9.1. Po tym, jak Alicja wczytała nową wartość, Robert też jej oczekuje.

Można dodatkowo doprecyzować diagram synchronizacji, aby pokazać, że każda operacja jest wprowadzana atomowo w pewnym punkcie w czasie. Bardziej skomplikowany przykład jest pokazany na rysunku 9.4 [10].

<sup>2</sup> Rejestr, w którym odczyty mogą zwracać albo starą, albo nową wartość, jeśli są wykonywane jednocześnie z zapisem, to *rejestr zwykły* (ang. *regular register*) [7, 25].



Rysunek 9.4. Wizualizowanie punktów w czasie, w których widoczny jest efekt odczytów i zapisów. Ostateczny odczyt klienta B nie jest liniowy

Na rysunku 9.4 obok odczytu i zapisu znajduje się trzeci rodzaj operacji:

- $\text{piu}(x, v_{\text{dawna}}, v_{\text{nowa}}) \Rightarrow r$  oznacza, że klient zażądał atomowej operacji *porównaj i ustaw* (zob. punkt „Porównaj i ustaw”). Jeśli aktualna wartość rejestru  $x$  to  $v_{\text{dawna}}$ , powinna zostać atomowo zmieniona na  $v_{\text{nowa}}$ . Jeżeli  $x \neq v_{\text{dawna}}$ , operacja zamiast zmieniać rejestr, powinna zwrócić błąd.  $r$  to odpowiedź bazy (OK lub błąd).

Dla każdej operacji na rysunku 9.4 dodano pionową kreskę (w prostokącie) reprezentującą czas, kiedy — jak uważamy — operacja została wykonana. Te kreski są połączone zgodnie z ich kolejnością, a efektem musi być poprawna sekwencja odczytów i zapisów wartości rejestru (każdy odczyt musi zwracać wartość ustawioną przez najnowszy zapis).

Wymóg liniowości polega na tym, że linie łączące kreski, które reprezentują operacje, zawsze muszą biec naprzód (od lewej do prawej). Nigdy nie mogą prowadzić do tyłu. Ten wymóg zapewnia opisaną wcześniej gwarancję nowości. Po zapisaniu lub wczytaniu nowej wartości we wszystkich późniejszych odczytach widoczna musi być zapisana wartość (do czasu jej ponownego nadpisania).

Na rysunku 9.4 warto zwrócić uwagę na kilka ciekawych szczegółów:

- Najpierw klient B wysyła żądanie odczytu  $x$ , potem klient D zgłasza żądanie ustawienia  $x$  na 0, a następnie klient A żąda ustawienia  $x$  na 1. Mimo to wartość zwracana klientowi B to 1 (jest to wartość zapisana przez A). Nie ma w tym nic złego. Oznacza to, że baza najpierw przetworzyła zapis klienta D, potem zapis klienta A, a na końcu odczyt klienta B. Choć nie jest to zgodne z kolejnością zgłaszania żądań, taki porządek jest akceptowalny, ponieważ trzy przesłane żądania są jednoczesne. Możliwe, że żądanie klienta B zostało opóźnione w sieci, dlatego dotarło do bazy dopiero po dwóch zapisach.
- Odczyt klienta B zwrócił 1 przed otrzymaniem przez klienta A odpowiedzi z bazy z informacją, że zapis wartości 1 się powiódł. Także to jest dopuszczalne. Nie oznacza to, że wartość została wczytana przed jej zapisaniem. Oznacza jednak, że odpowiedź OK od bazy dla klienta A została opóźniona w sieci.

- W tym modelu nie zakłada się izolacji transakcji. Inny klient może w dowolnym momencie zmodyfikować wartość. Na przykład C najpierw wczytuje 1, a potem 2, ponieważ między tymi dwoma odczytami wartość została zmieniona przez B. Można zastosować atomową operację „porównaj i ustaw” (*piu*) do sprawdzenia, czy wartość nie została jednocześnie zmieniona przez innego klienta. Żądania *piu* klientów B i C kończą się powodzeniem, jednak żądanie *piu* klienta D jest nieudane (w momencie przetwarzania go przez bazę wartość  $x$  nie wynosi już 0).
- Końcowy odczyt B (w ciemniejszym prostokącie) nie jest liniowy. Ta operacja jest jednocześnie z zapisem *piu* klienta C, który zmienia wartość  $x$  z 2 na 4. Gdyby nie inne żądania, zwrócenie 2 w odczycie klienta B byłoby akceptowalne. Jednak klient A wczytał już nową wartość (4) przed rozpoczęciem odczytu klienta B. Dlatego B nie może wczytać starszej wartości, niż zrobił to klient A. Jest to ta sama sytuacja co z Alicją i Robertem z rysunku 9.1.

Tak wygląda intuicyjne omówienie liniowości. Formalna definicja [6] opisuje ją bardziej precyzyjnie. Można przetestować (choć jest to kosztowne obliczeniowo), czy działanie systemu jest liniowe, rejestrując czas wszystkich żądań i odpowiedzi oraz sprawdzając, czy operacje można uporządkować sekwencyjnie w poprawnej kolejności [11].

## Liniowość a sekwencyjność

Liniowość można łatwo pomylić z sekwencyjnością (zob. punkt „Sekwencyjność”), ponieważ oba określenia oznaczają w przybliżeniu „możliwość uporządkowania w porządku sekwencyjnym”. Są to jednak odmienne gwarancje i ważne jest, by odróżniać je od siebie:

### Sekwencyjność

Sekwencyjność to poziom izolacji *transakcji*, na którym każda transakcja może wczytywać i zapisywać wiele obiektów (wierszy, dokumentów, rekordów); zob. punkt „Operacje na pojedynczych obiektach i na wielu obiektach”. Ten poziom gwarantuje, że transakcje działają tak, jakby zostały wykonane w *jakiś* porządku sekwencyjnym (gdzie każda transakcja jest wykonywana w całości przed rozpoczęciem następnej). Dopuszczalne jest, by ten porządek był inny od rzeczywistej kolejności uruchamiania transakcji [12].

### Liniowość

Liniowość to gwarancja nowości odczytów i zapisów rejestru (*jednego obiektu*). Tu operacje nie są grupowane w transakcje, dlatego ta technika nie zapobiega problemom takim jak zapis zniekształcający (zob. punkt „Zapis zniekształcający i fantomy”), chyba że zastosujesz dodatkowe techniki takie jak materializowanie konfliktów (zob. punkt „Materializowanie konfliktów”).

Baza może zapewniać zarówno sekwencyjność, jak i liniowość. To połączenie to *ściśła sekwencyjność* (ang. *strict serializability*) lub *silna sekwencyjność z jedną kopią* (ang. *strong one-copy serializability*) [4, 13]. Implementacje sekwencyjności z wykorzystaniem blokad dwuetapowych (zob. punkt „Blokady dwuetapowe”) lub rzeczywistego sekwencyjnego wykonywania (zob. punkt „Rzeczywiste wykonywanie sekwencyjne”) są zwykle liniowe.

Jednak sekwencyjna izolacja snapshotów (zob. punkt „Sekwencyjna izolacja snapshotów”) nie jest liniowa. W tym podejściu z zasady odczyt odbywa się ze spójnego snapshota, aby uniknąć rywalizacji o blokady między jednostkami wczytującymi i zapisującymi. Cały sens stosowania spójnych snapshotów polega na tym, że nie obejmują one zapisów nowszych niż snapshot, dlatego odczyty ze snapshota nie są liniowe.

## Poleganie na liniowości

W jakich okolicznościach liniowość jest przydatna? Wyświetlanie końcowego wyniku meczu piłkarskiego to blahy przykład. Wynik nieaktualny o kilka sekund zapewne nie spowoduje żadnych szkód. Jednak w kilku obszarach liniowość jest ważnym wymogiem potrzebnym do poprawnego działania systemu.

### Blokady i wybór lidera

System z replikacją z jednym liderem musi gwarantować, że rzeczywiście występuje tylko jeden lider, a nie kilku (co skutkuje „rozszczeniem mózgu”). Jednym ze sposobów wyboru lidera jest zastosowanie blokady. Każdy uruchamiany węzeł próbuje zająć blokadę, a ten, któremu się to powiedzie, zostaje liderem [14]. Niezależnie od tego, jak blokada jest zaimplementowana, musi być liniowa. Wszystkie węzły muszą się zgadzać co do tego, który węzeł ma blokadę. W przeciwnym razie to rozwiązanie jest bezużyteczne.

Do implementowania blokad rozproszonych i procesu wyboru lidera często używane są usługi zapewniające koordynację, np. Apache ZooKeeper [15] i etcd [16]. Te usługi używają algorytmów osiągania konsensusu do implementowania operacji liniowych w sposób odporny na błędy (takie algorytmy są opisane dalej w rozdziale, w punkcie „Konsensus przy zachowaniu odporności na błędy”)³. Jednak poprawne zaimplementowanie wyboru lidera i blokad nadal wymaga uwzględnienia wielu szczegółów (zob. np. problem odgradzania w punkcie „Lider i blokady”). Pomocne są tu biblioteki takie jak Apache Curator [17], które zapewniają wysokopoziomowe receptury oparte na ZooKeeperze, ale główną podstawą zadań z obszaru koordynacji jest usługa liniowego składowania danych.

Blokady rozproszone w niektórych bazach rozproszonych (np. w bazie Real Application Clusters — RAC — firmy Oracle) są stosowane w dużo bardziej precyzyjny sposób. W bazie RAC używane są blokady stron dysku, a do tego samego dyskowego systemu składowania danych dostęp ma wiele węzłów. Ponieważ te liniowe blokady dotyczą ścieżki krytycznej w transakcji, w systemach z bazą RAC do komunikacji między węzłami bazy zwykle używana jest specjalnie do tego przeznaczona sieć typu *cluster interconnect*.

### Więzy i gwarancje unikatowości

Więzy unikatowości są często stosowane w bazach. Na przykład nazwa użytkownika lub adres e-mail muszą być unikatowym identyfikatorem użytkownika, a w usłudze składowania danych w plikach nie mogą występować dwa pliki o tej samej ścieżce i nazwie. Jeśli chcesz wymuszać te więzy na etapie zapisu danych (tak aby w sytuacji, gdy dwie osoby chcą jednocześnie utworzyć konto lub plik o tej samej nazwie, jedna z prób zakończyła się błędem), potrzebujesz liniowości.

Ten scenariusz jest podobny jak przy stosowaniu blokady. Gdy użytkownik rejestruje się w usłudze, można to porównać z zajęciem „blokady” wybranej nazwy użytkownika. Ta operacja bardzo

---

<sup>3</sup> Ścisłe rzecz biorąc, ZooKeeper i etcd zapewniają liniowe zapisy, jednak odczyty mogą być nieaktualne, ponieważ domyślnie mogą być obsługiwane przez dowolną replikę. Możesz opcjonalnie zażądać liniowych odczytów. W etcd ten model jest nazywany *odczytem z użyciem kworum* [16], a w ZooKeeperze przed odczytem należy wywołać funkcję `sync()` [15]. Zob. „Implementowanie baz liniowych za pomocą rozgłaszania z uporządkowaniem całkowitym”.

przypomina też atomowe „porównywanie i ustawianie” — jako identyfikator użytkownika stosuje się nazwę, którą ta osoba wybrała, pod warunkiem jednak że dana nazwa nie jest już zajęta.

Podobne problemy pojawiają się, jeśli chcesz zagwarantować, że stan konta bankowego nigdy nie będzie ujemny, że nie sprzedasz więcej produktów, niż masz w magazynie, lub że dwie osoby nie zarezerwują jednocześnie tego samego miejsca w samolocie lub teatrze. Wszystkie te ograniczenia wymagają jednej aktualnej wartości (stanu konta, stanu magazynu, wolnych miejsc), co do której zgadzają się wszystkie węzły.

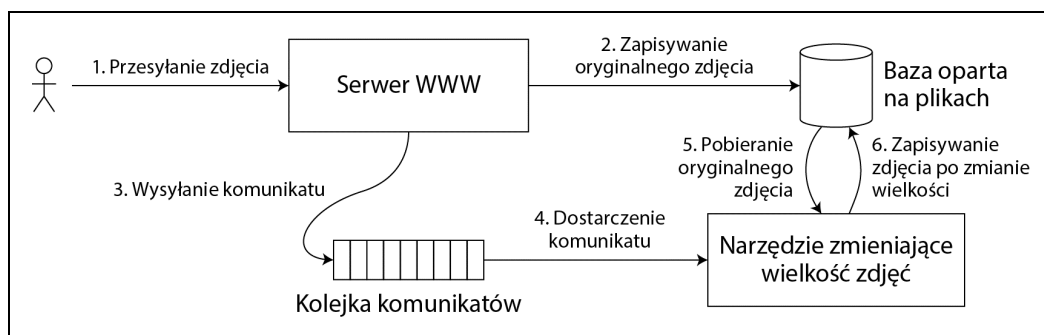
W rzeczywistych aplikacjach czasem można luźno traktować takie ograniczenia (np. jeśli sprzedano za dużo biletów na samolot, można przesunąć pasażerów na inny lot i zaproponować im rekompensatę za niedogodności). W takich sytuacjach liniowość może nie być konieczna. Tego rodzaju luźno traktowane ograniczenia są opisane w punkcie „Aktualność i integralność”.

Jednak sztywne więzy unikatowości, np. takie, jakie zwykle występują w bazach relacyjnych, wymagają liniowości. Inne rodzaje więzów, choćby klucze zewnętrzne lub więzy dotyczące atrybutów, można implementować bez liniowości [19].

### Zależności czasowe między kanałami

Zwróć uwagę na pewien szczegół na rysunku 9.1. Gdyby Alicja nie wykrzyknęła wyniku, Robert nie wiedziałby, że odpowiedź na jego zapytanie jest nieaktualna. Po kilku sekundach odświeżyłby stronę i ostatecznie zobaczył końcowy wynik. Naruszenie liniowości zostało zauważone tylko z powodu dodatkowego kanału komunikacji w systemie (głosu Alicji docierającego do uszu Roberta).

Podobne sytuacje mogą zaistnieć w systemach komputerowych. Wyobraź sobie witrynę, w której użytkownicy mogą przysyłać zdjęcia, a działający w tle proces zmniejsza rozdzielczość zdjęć, aby przyspieszyć ich pobieranie (w postaci miniatur). Architektura i przepływ danych w tym systemie są przedstawione na rysunku 9.5.



Rysunek 9.5. Serwer WWW i narzędzie zmieniające wielkość zdjęć komunikują się z użyciem opartej na plikach usługi przechowywania danych i kolejki komunikatów, co grozi sytuacją wyścigu

Narzędzie zmieniające rozmiar zdjęć musi otrzymać bezpośrednie instrukcje wykonania swojego zadania. Te interfejsy są przysyłane z serwera WWW do narzędzia za pomocą kolejki komunikatów (zob. rozdział 11.). Serwer WWW nie umieszcza w kolejce całego zdjęcia, ponieważ większość brokerów komunikatów jest zaprojektowana z myślą o niewielkich komunikatach, a zdjęcie może zajmować kilka megabajtów. Zamiast tego zdjęcie jest najpierw zapisywane w opartej na plikach



usłudze składowania danych, a po zakończeniu zapisu instrukcja dla narzędzia zmieniającego wielkość fotografii jest umieszczana w kolejce.

Jeśli oparta na plikach usługa przechowywania danych jest liniowa, ten system powinien działać poprawnie. W przeciwnym razie istnieje ryzyko sytuacji wyścigu. Kolejka komunikatów (kroki 3. i 4. na rysunku 9.5) może działać szybciej niż wewnętrzna replikacja w usłudze przechowywania danych. Jeśli tak jest, to gdy narzędzie pobierze zdjęcie (krok 5.), może zobaczyć jego starą wersję lub nie znaleźć żadnych danych. Jeżeli narzędzie przetworzy dawną wersję zdjęcia, oryginalna fotografia i miniatura w usłudze przechowywania danych będą trwale niespójne.

Ten problem pojawia się, ponieważ istnieją dwa różne kanały komunikacji między serwerem WWW a narzędziem modyfikującym zdjęcia: usługa przechowywania danych i kolejka komunikatów. Bez zapewnianych dzięki liniowości gwarancji nowości możliwa jest sytuacja wyścigu między tymi dwoma kanałami. Jest to scenariusz podobny jak na rysunku 9.1, gdzie także widoczna jest sytuacja wyścigu między dwoma kanałami komunikacji: replikacją w bazie i fizycznym kanałem dźwiękowym między ustami Alicji i uszami Roberta.

Liniowość nie jest jedynym sposobem unikania sytuacji wyścigu, ale stanowi rozwiązanie najprostsze do zrozumienia. Jeśli kontrolujesz dodatkowy kanał komunikacji (dzieje się tak w przypadku kolejki komunikatów, ale już nie w scenariuszu z Alicją i Robertem), możesz zastosować inne podejście podobne do tego z punktu „Odczyt własnych zapisów”, choć powoduje to wzrost złożoności systemu.

## Implementowanie systemów liniowych

Po przyjrzeniu się kilku przykładom, w których liniowość jest przydatna, warto się zastanowić nad tym, jak zaimplementować zapewniający ją system.

Ponieważ liniowość w istocie oznacza „działanie tak, jakby istniała tylko jedna kopia danych, a wszystkie operacje na niej były atomowe”, najprostszym rozwiązaniem jest zastosowanie jednej kopii danych. Jednak to podejście nie zapewnia odporności na błędy. Jeśli węzeł przechowujący tę jedną kopię zawiedzie, dane zostaną utracone — a przynajmniej staną się niedostępne do czasu wznowienia pracy węzła.

Najczęściej stosowana technika zapewniania odporności systemu na błędy to replikacja. Wróćmy do metod replikacji opisanych w rozdziale 5. i sprawdźmy, czy pozwalają zapewnić liniowość:

### *Replikacja z jednym liderem (potencjalnie liniowa)*

W systemie z replikacją z jednym liderem (zob. punkt „Liderzy i obserwatorzy”) lider obejmuje główną kopię danych używaną do zapisów, a obserwatorzy przechowują kopie zapasowe danych w innych węzłach. Jeśli odczyty korzystają z lidera lub z synchronicznie aktualizowanych obserwatorów, to rozwiązanie jest *potencjalnie* liniowe<sup>4</sup>. Jednak nie każda baza z jednym liderem rzeczywiście jest liniowa. Wynika to albo z projektu (np. z powodu używania izolacji snapshots), albo z błędów związanych z jednoczesnością [10].

<sup>4</sup> Podział bazy z jednym liderem na partycje, tak że dla każdej partycji używany jest odrębny lider, nie wpływa na liniowość, ponieważ zapewnia ona tylko gwarancje dla jednego obiektu. Inną kwestią są transakcje dotyczące różnych partycji (zob. punkt „Transakcje rozproszone a konsensus”).

Odczyty z użyciem lidera wymagają przyjęcia założenia, że na pewno wiadomo, kto jest liderem. W punkcie „Prawda jest definiowana przez większość” opisano, że węzeł może błędnie zakładać, iż jest liderem. Jeśli taki lider ciągle obsługuje żądania, prawdopodobnie naruszy liniowość [20]. Gdy stosuje się replikację asynchroniczną, przełączanie awaryjne może nawet prowadzić do utraty zatwierdzonych zapisów (zob. punkt „Radzenie sobie z przestojami węzłów”), co narusza zarówno trwałość, jak i liniowość.

#### *Algorytmy wykorzystujące konsensus (liniowe)*

Niektóre algorytmy osiągania konsensusu (opisane dalej w tym rozdziale) przypominają replikację z jednym liderem. Jednak protokoły osiągania konsensusu obejmują mechanizmy zapobiegania „rozzczępieniu mózgu” i nieaktualnym replikom. Dzięki temu takie algorytmy pozwalają bezpiecznie implementować bazy zapewniające liniowość. W ten sposób działają np. ZooKeeper [21] i etcd [22].

#### *Replikacja z wieloma liderami (nieliniowa)*

Systemy z replikacją z wieloma liderami zwykle nie są liniowe, ponieważ jednocześnie przetwarzają zapisy w wielu węzłach i asynchronicznie replikują je w innych węzłach. Dlatego mogą wykonywać sprzeczne zapisy wymagające rozwiązania konfliktu (zob. punkt „Radzenie sobie z konfliktami przy zapisie”). Takie konflikty to skutek braku jednej kopii danych.

#### *Replikacja bez lidera (prawdopodobnie nieliniowa)*

Jeśli chodzi o systemy z replikacją bez lidera (w stylu bazy Dynamo; zob. punkt „Replikacja bez lidera”), niektórzy twierdzą, że można uzyskać „silną spójność” dzięki wymogowi odczytów i zapisów na podstawie kworum ( $w + r > n$ ). W zależności od dokładnej konfiguracji kworum (i od tego, jak zdefiniowana jest silna spójność) nie jest to do końca prawdą.

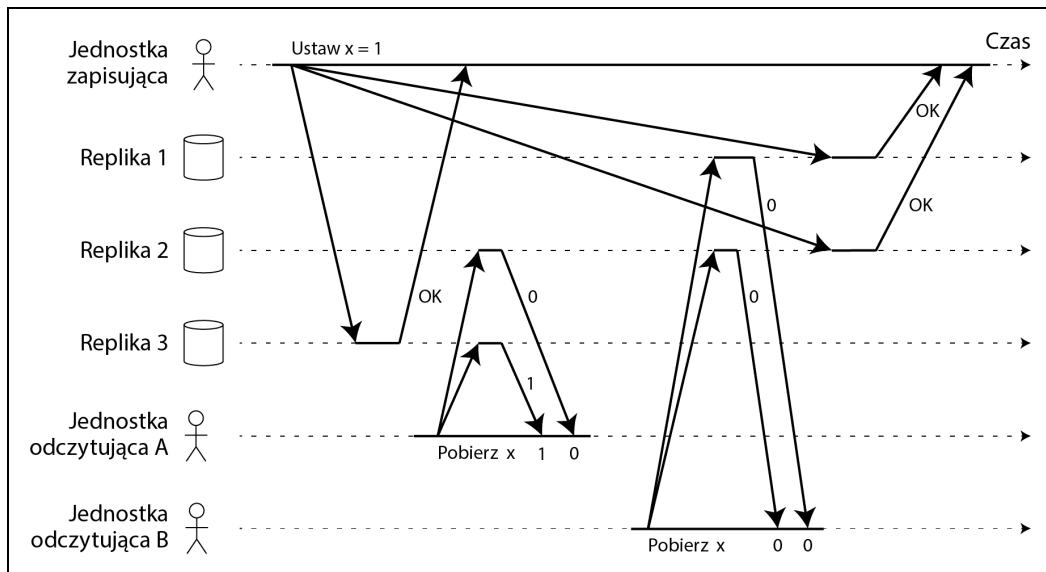
Metody rozwiązywania konfliktów „ostatni zapis wygrywa” oparte na zegarach czasu rzeczywistego (np. takie jak w Cassandra; zob. punkt „Poleganie na zegarach synchronizowanych”) prawie na pewno nie są liniowe, ponieważ z powodu odchyłeń zegarów nie można zagwarantować, że znaczniki czasu będą zgodne z rzeczywistą kolejnością zdarzeń. Kwora niepełne („Kwora niepełne i mechanizm hinted-handoff”) także uniemożliwiają liniowość. Nawet jeśli występują ścisłe kwora, możliwe są działania nieliniowe, co wykazano w następnym punkcie.

### **Liniowość a kwora**

Intuicyjnie wydaje się, że odczyty i zapisy przy stosowaniu ścisłego kworum powinny być liniowe (tak jak w bazie Dynamo). Jeśli jednak występują zmienne opóźnienia sieciowe, możliwa jest sytuacja wyścigu, co ilustruje rysunek 9.6.

Na rysunku 9.6 wartość początkowa  $x$  to 0, a klient zapisujący dane zmienia wartość  $x$  na 1, przysyłając zapis do wszystkich trzech replik ( $n = 3$ ,  $w = 3$ ). Jednocześnie klient A wczytuje dane z kworum dwóch węzłów ( $r = 2$ ) i widzi w jednym z nich nową wartość 1. Także jednocześnie z zapisem klient B wczytuje wartość z innego kworum dwóch węzłów i pobiera z obu dawną wartość, 0.

Warunek uzyskania kworum jest spełniony ( $w + r > n$ ), jednak mimo to system nie działa liniowo. Żądanie od B zaczyna się po zakończeniu obsługi żądania od A. Jednak B otrzymuje dawną wartość, choć A pobrał nowe dane. Ponownie jest to sytuacja taka jak z Alicją i Robertem na rysunku 9.1.



Rysunek 9.6. Nieliniowe działanie mimo kworum ścisłego

Co ciekawe, *możliwe* jest zapewnienie liniowości kworum w stylu bazy Dynamo, choć kosztem spadku wydajności. Jednostka odczytująca musi synchronicznie wykonywać uspójnianie przy odczycie (zob. punkt „Uspójnianie przy odczycie i eliminowanie entropii”), zanim wyniki zostaną zwrócone do aplikacji [23], a jednostka zapisująca przed przesłaniem zapisów musi wczytywać najnowszy stan z kworum węzłów [24, 25]. Riak z powodu spadku wydajności nie przeprowadza synchronicznego uspójniania przy odczycie [26]. Cassandra *odczeka* na zakończenie uspójniania przy odczycie danych z kworum [27], jednak nie działa liniowo, gdy wykonywanych jest wiele jednoczesnych zapisów tego samego klucza (ponieważ do rozwiązywania konfliktów używa się metody ostatni zapis wygrywa).

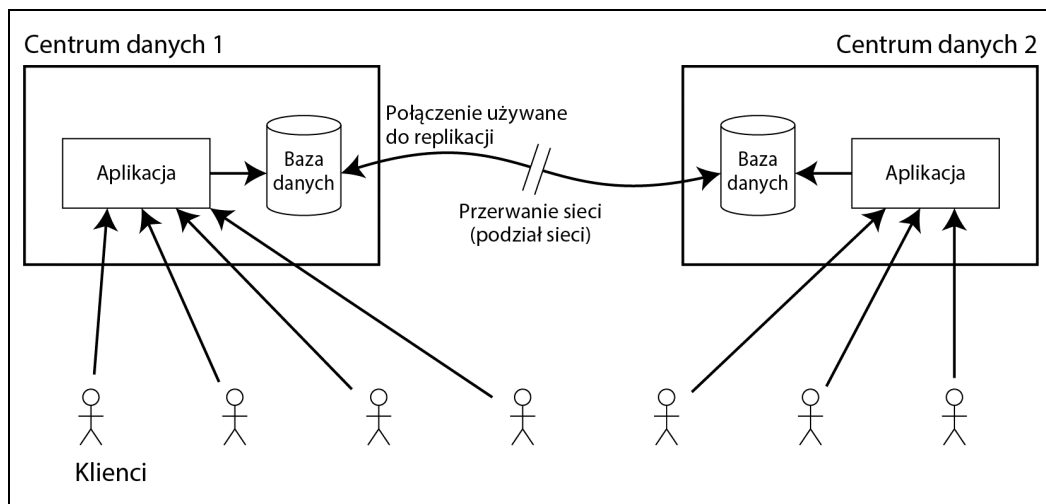
Ponadto w ten sposób można zaimplementować tylko liniowe operacje odczytu i zapisu. Nie jest to możliwe dla operacji „porównaj i ustaw”, ponieważ wymaga ona algorytmu osiągnięcia konsensusu [28].

W podsumowaniu można napisać, że najbezpieczniej jest przyjąć, iż system bez lidera z replikacją w stylu z bazy Dynamo nie zapewnia liniowości.

## Koszty liniowości

Ponieważ niektóre metody replikacji mogą zapewniać liniowość, a inne nie, warto dobrze się zastanowić nad wadami i zaletami liniowości.

Niektóre zastosowania różnych metod replikacji zostały już opisane w rozdziale 5. Pokazano np., że replikacja z wieloma liderami jest często dobrym wyborem dla replikacji na poziomie wielu centrów danych (zob. punkt „Eksploracja wielu centrów danych”). Przykład takiego rozwiązania został pokazany na rysunku 9.7.



Rysunek 9.7. Przerwanie sieci wymusza wybór między zachowaniem liniowości lub dostępności

Pomyśl, co się stanie, jeśli nastąpi przerwanie sieci między dwoma centrami danych. Załóżmy, że sieć w każdym centrum danych działa, a klienci mogą się skontaktować z centrami, jednak centra nie mogą się połączyć ze sobą nawzajem.

W bazie z wieloma liderami każde centrum danych może działać w normalny sposób. Ponieważ zapisy z jednego centrum są asynchronicznie replikowane w innym, trafiają do kolejki i są przesyłane po przywróceniu pracy sieci.

Jeśli jednak stosuje się replikację z jednym liderem, lider musi się znajdować w jednym z centrów danych. Trzeba, aby wszystkie zapisy i liniowe odczyty były obsługiwane przez tego lidera. Dlatego klienci połączone z centrum pełniącym funkcję obserwatora wymagają synchronicznego przekazywania żądań odczytu i zapisu siecią do centrum lidera.

Gdy w konfiguracji z jednym liderem w sieci łączącej centra danych wystąpią zakłócenia, klienci połączone z centrami pełniącymi funkcję obserwatora nie będą się mogły skontaktować z liderem. Nie będą więc mogły wykonywać zapisów w bazie ani liniowych odczytów. Nadal mogą wczytywać dane z obserwatorów, jednak te dane mogą być nieaktualne (nieliniowe). Jeśli aplikacja wymaga liniowych odczytów i zapisów, zakłócenie pracy sieci powoduje, że w centrach danych, które nie są w stanie skontaktować się z liderem, aplikacja będzie niedostępna.

Dla klientów mogących bezpośrednio połączyć się z centrum danych, w którym działa lider, problem nie występuje, ponieważ aplikacja działa dla nich normalnie. Jednak klienci mające dostęp tylko do centrów danych z obserwatorami nie będą mogli używać aplikacji do czasu naprawienia połączenia sieciowego.

## Twierdzenie CAP

Opisany problem nie jest wyłącznie wynikiem stosowania replikacji z jednym liderem lub z wieloma liderami. Takie trudności występują w każdej liniowej bazie niezależnie od implementacji.

Problem nie ogranicza się też do systemów z wieloma centrami danych — może wystąpić zawsze, gdy sieć jest zawodna (także w ramach jednego centrum danych). Oto możliwe kompromisy<sup>5</sup>:

- Jeśli aplikacja *wymaga* liniowości, a niektóre repliki są oddzielone od innych z powodu problemów z siecią, część replik nie może przetwarzać żądań, gdy połączenie nie działa. Muszą albo poczekać na naprawienie sieci, albo zwrócić błąd (w obu sytuacjach stają się *niedostępne*).
- Jeżeli aplikacja *nie wymaga* liniowości, można ją napisać w taki sposób, by każda replika mogła niezależnie przetwarzać żądania nawet gdy nie ma połączenia z innymi replikami (zastosuj np. model z wieloma liderami). Wtedy aplikacja pozostaje *dostępna* mimo problemów z siecią, jednak nie działa liniowo.

Tak więc aplikacja, które nie wymaga liniowości, może być odporniejsza na problemy z siecią. To spostrzeżenie jest nazywane *twierdzeniem CAP* [29, 30, 31, 32], które to określenie wymyślił w 2000 r. Eric Brewer, jednak opisane kompromisy są znane projektantom rozproszonych baz danych od lat 70. [33, 34, 35, 36].

Twierdzenie CAP zostało pierwotnie zaproponowane jako prosta reguła bez precyzyjnych definicji i miało pomóc w rozpoczęciu dyskusji na temat kompromisów w bazach danych. W momencie powstania tego twierdzenia twórcy wielu rozproszonych baz koncentrowali się na zapewnianiu liniowości w klastrze maszyn ze współużytkowanym magazynem danych [18], a twierdzenie CAP zachęcało inżynierów do analizy większej przestrzeni projektowej systemów rozproszonych bez zasobów współużytkowanych, które lepiej nadawały się do implementowania usług sieciowych działających w dużej skali [37]. Twierdzeniu CAP zawdzięczamy zmianę podejścia i eksplozję nowych technologii bazodanowych (nazywanych NoSQL) widoczną od połowy pierwszej dekady XXI w.

## Mało pomocne twierdzenie CAP

CAP jest czasem przedstawiane tak: *spójność, dostępność, odporność na podział: wybierz 2 z 3*. Niestety, takie ujęcie jest mylące [32], ponieważ podział sieci jest rodzajem błędu, dlatego nie mamy w tej kwestii wyboru. Takie błędy będą się zdarzać niezależnie od tego, czy tego chcesz [38].

Gdy sieć działa prawidłowo, system może zapewniać zarówno spójność (liniowość), jak i pełną dostępność. Po wystąpieniu błędu sieci trzeba wybrać między liniowością a pełną dostępnością. Dlatego lepszym sposobem na opis twierdzenia CAP jest: *albo spójny, albo dostępny po podziale* [39]. Bardziej niezawodne sieci wymagają rzadszego dokonywania tego wyboru, jednak w pewnym momencie jest on nieunikniony.

W dyskusjach na temat CAP pojawiają się różne sprzeczne definicje *dostępności*, a formalne ujęcie tego twierdzenia [30] nie jest zgodne z jego standardowym znaczeniem [40]. Wiele tzw. wysoce dostępnych (odpornych na błędy) systemów nie jest zgodnych ze specyficzną definicją dostępności z twierdzenia CAP. Ostatecznie wokół twierdzenia CAP narosło wiele nieporozumień i wątpliwości, przez co nie pomaga ono lepiej zrozumieć systemów. Dlatego najlepiej jest unikać twierdzenia CAP.

<sup>5</sup> Te dwie możliwości są czasem nazywane CP (spójne, ale niedostępne po podziale sieci) i AP (dostępne, ale niespójne po podziale sieci). Jednak ten sposób kategoryzacji ma kilka wad [9], dlatego najlepiej jest go unikać.

Formalna definicja twierdzenia CAP [30] ma bardzo wąski zakres. Uwzględnia tylko jeden model spójności (liniowość) i jeden rodzaj błędów (*podział sieci*<sup>6</sup>, czyli rozdzielenie aktywnych węzłów). W ogóle nie obejmuje opóźnień sieciowych, niesprawnych węzłów lub kompromisów. Dlatego choć twierdzenie CAP było historycznie istotne, ma niewielką praktyczną wartość w obszarze projektowania systemów [9, 40].

W systemach rozproszonych występuje o wiele więcej ciekawych dowodów na niemożliwość [41], a twierdzenie CAP zostało obecnie zastąpione bardziej precyzyjnymi dowodami [2, 42], dlatego dziś ma głównie znaczenie historyczne.

### Liniowość a opóźnienia sieciowe

Choć liniowość to przydatna gwarancja, w praktyce zaskakująco nieliczne systemy są liniowe. Na przykład nawet pamięć RAM w nowych procesorach wielordzeniowych nie jest liniowa [43]. Jeśli wątek działający w jednym rdzeniu procesora zapisuje dane pod jednym adresem w pamięci, a wątek z innego rdzenia zaraz potem wczytuje dane z tego adresu, nie ma gwarancji, że wczytana zostanie wartość zapisana przez pierwszy wątek (chyba że używana jest *bariera w pamięci* lub *odgrozdzenie* [44]).

Takie działanie wynika z tego, że każdy rdzeń procesora ma własną pamięć podręczną i bufor danych. Dostęp do pamięci domyślnie odbywa się najpierw z użyciem pamięci podręcznej, a wszystkie zmiany są asynchronicznie zapisywane w pamięci głównej. Ponieważ dostęp do danych z pamięci podręcznej odbywa się znacznie szybciej niż do danych z pamięci głównej [45], to rozwiązanie jest konieczne do uzyskania wysokiej wydajności w nowoczesnych procesorach. Jednak powstaje wtedy kilka kopii danych (jedna w pamięci głównej i prawdopodobnie kilka innych w różnych pamięciach podręcznych), które są aktualizowane asynchronicznie, co powoduje utratę liniowości.

Dlaczego zdecydowano się na ten kompromis? Nie ma sensu posługiwać się twierdzeniem CAP do uzasadnienia modelu spójności pamięci w środowisku z wieloma rdzeniami. Zwykle przyjmuje się, że komunikacja w ramach jednego komputera działa niezawodnie. Nie oczekujemy, że jeden rdzeń procesora będzie potrafił funkcjonować normalnie po odłączeniu od reszty komputera. Dlatego powodem rezygnacji z liniowości jest *wydajność*, a nie odporność na błędy.

To samo dotyczy wielu rozproszonych baz, w których zdecydowano się nie zapewniać gwarancji liniowości. Zrobiono to głównie w celu poprawy wydajności, a nie po to, by uzyskać odporność na błędy [46]. Liniowość skutkuje powolną pracą i to w każdym momencie, a nie tylko po awarii sieci.

Czy nie da się wymyślić wydajniejszej implementacji baz liniowych? Wydaje się, że jest to niewykonalne. Attiya i Welch [47] dowodzą, że jeśli zależy Ci na liniowości, czas odpowiedzi dla żądań odczytu i zapisu jest co najmniej proporcjonalny do wahania opóźnień w sieci. W sieci z bardzo zmiennymi opóźnieniami, czyli w większości sieci komputerowych (zob. punkt „Limity czasu i nieograniczone opóźnienia”), czas odpowiedzi liniowych odczytów i zapisów z natury będzie długi. Nie istnieje szybszy algorytm zapewniania liniowości, jednak słabsze modele spójności pozwalają

---

<sup>6</sup> Zgodnie z opisem z punktu „Błędy sieci w praktyce” w tej książce określenie *podział* (ang. *partitioning*) oznacza celowe rozbięcie dużego zbioru danych na mniejsze (inna nazwa to *sharding*; zob. rozdział 6.). Natomiast podział sieci to określony typ błędu, który zwykle nie jest rozważany niezależnie od problemów innego rodzaju. Jednak ponieważ słowo *partitioning* to P w nazwie CAP, nie da się tu uniknąć niejasności.

osiągnąć znacznie wyższą wydajność. Dlatego omawiany kompromis jest ważny w systemach wrażliwych na wysokie opóźnienia. W rozdziale 12. opisano techniki rezygnacji z liniowości bez poświęcania poprawności.

## Gwarancje uporządkowania

Wcześniej wspomniano, że rejestr liniowy działa tak, jakby istniała tylko jedna kopia danych. Wygląda to tak, jakby każda operacja była wykonywana atomowo w określonym momencie. Z tej definicji wynika, że operacje są wykonywane w ściśle zdefiniowanym porządku. Na rysunku 9.4 ten porządek przedstawiono, łącząc operacje zgodnie z kolejnością, w jakiej wydają się wykonywane.

Kolejność operacji to temat powtarzający się w tej książce, co sugeruje, że może to być ważne, fundamentalne zagadnienie. Przypomnijmy pokrótce kilka innych kontekstów, w jakich omawiano uporządkowanie:

- W rozdziale 5. pokazano, że głównym celem lidera w replikacji z jednym liderem jest określenie *kolejności zapisów* w dzienniku replikacji, czyli porządku, w jakim obserwatorzy wprowadzają te zapisy. Jeśli nie ma jednego lidera, mogą wystąpić konflikty z powodu jednoczesnych operacji (zob. punkt „Radzenie sobie z konfliktami przy zapisie”).
- Sekwencyjność (omawiana w rozdziale 7.) ma zapewniać, że transakcje działają tak, jakby były wykonywane w *określonym porządku sekwencyjnym*. Ten efekt można osiągnąć, wykonując transakcje dosłownie w porządku sekwencyjnym lub dopuszczając jednoczesne wykonywanie operacji i zapobieganie przy tym konfliktom (za pomocą blokad lub anulowania działań).
- Stosowanie w systemach rozproszonych znaczników czasu i zegarów, co omówiono w rozdziale 8. (zob. punkt „Poleganie na zegarach synchronizowanych”), to następna próba wprowadzenia porządku w nieuporządkowanym świecie — np. w celu określenia, który z dwóch zapisów nastąpił później.

Okazuje się, że istnieją ściśle powiązania między uporządkowaniem, liniowością i konsensusem. Choć to zagadnienie jest bardziej teoretyczne i abstrakcyjne niż reszta książki, okazuje się bardzo pomocne do zrozumienia, co system potrafi, a czego nie potrafi zrobić. Ten temat omówiono w paru następnych punktach.

## Uporządkowanie i przyczynowość

Jest kilka powodów, dla których temat uporządkowania wciąż powraca. Jednym z nich jest to, że pomaga ono zachować *przyczynowość*. W tej książce zobaczyłeś już kilka przykładów, w których przyczynowość miała znaczenie:

- W punkcie „Odczyty ze spójnym przedrostkiem” widziałeś przykład, w którym obserwator konwersacji najpierw widzi odpowiedź, a dopiero potem samo pytanie. Jest to mylące, ponieważ narusza intuicyjne zrozumienie zasady przyczyny i skutku. Jeśli pojawiła się odpowiedź, z pewnością najpierw musiało zostać zadane pytanie, ponieważ osoba udzielająca odpowiedzi musiała zobaczyć to pytanie (przy założeniu, że nie ma zdolności paranormalnych i nie widzi przyszłości). Mówimy, że między pytaniem a odpowiedzią zachodzi *związek przyczynowy*.

- Podobny wzorec pojawił się na rysunku 5.9, gdzie pokazano replikację między trzema liderami i podkreślono, że niektóre zapisy mogą „wyprzedzić” inne z powodu opóźnień sieciowych. Z perspektywy jednej z replik wygląda to tak, jakby pojawiła się aktualizacja wiersza, który jeszcze nie istnieje. Przyczynowość oznacza tu, że wiersz najpierw trzeba utworzyć, a dopiero później można go aktualizować.
- W punkcie „Wykrywanie jednoczesnych zapisów” zauważono, że jeśli są dwie operacje A i B, występują trzy możliwości: albo A została wykonana przed B, albo B przed A, albo A i B są jednoczesne. Relacja *wystąpiło przed* to następny przykład przyczynowości. Jeśli A wystąpiła przed B, to oznacza, że B mogła wiedzieć o A, być zbudowana na podstawie A lub zależeć od A. Gdy A i B są jednoczesne, nie występuje związek przyczynowy między nimi. Oznacza to, że mamy pewność, iż żadna z nich nie wiedziała o drugiej.
- W kontekście izolacji snapshotów na potrzeby transakcji („Izolacja snapshotów i powtarzalny odczyt”) stwierdzono, że transakcja wczytuje dane ze spójnego snapshota. Co jednak oznacza „spójność” w tym kontekście? Chodzi o *spójność przyczynową*. Jeśli snapshot obejmuje odpowiedź, musi też zawierać zadane pytanie [48]. Obraz całej bazy z danego momentu zapewnia, że jest ona spójna przyczynowo. Widoczne są efekty wszystkich operacji, które się wydarzyły przed tym momentem, a skutki późniejszych zdarzeń nie są dostępne. Zapis zniekształcający (przedstawione na rysunku 7.6 odczyty niepowtarzalne) oznacza wczytywanie danych w stanie niezgodnym z przyczynowością.
- Związek przyczynowy pokazano też w przykładach zapisu zniekształcającego przez różne transakcje (zob. punkt „Zapis zniekształcający i fantomy”). Na rysunku 7.8 Alicja mogła zrezygnować z dyżuru, ponieważ transakcja uznała, że Robert wciąż jest dyżurnym (i na odwrót). W tej sytuacji rezygnacja z dyżuru jest zależna przyczynowo od sprawdzania, kto aktualnie jest na niego zapisany. Algorytm SSI (zob. punkt „Algorytm SSI”) wykrywa zapis zniekształcający, analizując związki przyczynowe między transakcjami.
- W przykładzie z oglądaniem meczu przez Alicję i Roberta (rysunek 9.1) to, że Robert otrzymał z serwera nieaktualne dane po usłyszeniu, jak Alicja wykrzykuje wynik, jest naruszeniem przyczynowości. Okrzyk Alicji jest przyczynowo zależny od ogłoszenia wyniku, dlatego także Robert powinien móc zobaczyć rezultat po usłyszeniu go od Alicji. Ten sam wzorec powtórzył się w punkcie „Zależności czasowe między kanałami” w formie usługi zmieniającej rozmiar zdjęć.

Przyczynowość wymusza uporządkowanie zdarzeń. Przyczyny występują przed skutkami; komunikat jest wysyłany, zanim zostanie odebrany, a pytanie pojawia się przed odpowiedzią. I, podobnie jak w rzeczywistym życiu, jedna rzecz prowadzi do drugiej. Jeden węzeł wczytuje dane, a następnie w efekcie tego coś zapisuje. Inny węzeł wczytuje zapisane dane, po czym zapisuje coś innego itd. Ten łańcuch przyczynowo zależnych operacji wyznacza porządek przyczynowy w systemie, czyli określa, co zdarzyło się wcześniej, a co później.

Jeśli system jest zgodny z uporządkowaniem wymuszonym przez przyczynowość, jest *spójny przyczynowo*. Na przykład izolacja snapshotów zapewnia spójność przyczynową. Gdy wczytujesz dane z bazy i widzisz jakieś dane, musisz też mieć dostęp do przyczynowo wcześniejszych danych (przy założeniu, że nie zostały one w międzyczasie usunięte).



## Uporządkowanie przyczynowe nie jest uporządkowaniem całkowitym

*Uporządkowanie całkowite* (ang. *total order*) umożliwia porównanie dwóch dowolnych elementów. Jeśli więc masz dwa elementy, zawsze możesz stwierdzić, który jest większy, a który mniejszy. Na przykład liczby naturalne są uporządkowane całkowicie. Jeśli podam dwie liczby, np. 5 i 13, potrafisz stwierdzić, że 13 jest większe niż 5.

Jednak zbiory matematyczne nie są uporządkowane całkowicie. Czy zbiór  $\{a, b\}$  jest większy niż  $\{b, c\}$ ? No cóż, nie można ich porównać, ponieważ żaden nie jest podzbiorem drugiego. Mówimy więc, że są *nieporównywalne*. Dlatego zbiory matematyczne są *uporządkowane częściowo*. W niektórych sytuacjach jeden zbiór jest większy od drugiego (gdy obejmuje wszystkie elementy tego drugiego), jednak w innych przypadkach zbiorów nie da się porównać.

Różnica między uporządkowaniem całkowitym a częściowym jest widoczna w różnych modelach spójności bazy danych:

### *Liniowość*

W systemie liniowym występuje *uporządkowanie całkowite* operacji. Jeśli system działa tak, jakby istniała tylko jedna kopia danych, i każda operacja jest atomowa, oznacza to, że dla dowolnych dwóch operacji zawsze można określić, która z nich została wykonana wcześniej. To uporządkowanie całkowite jest przedstawione jako oś czasu na rysunku 9.4.

### *Przyczynowość*

Mówimy, że dwie operacje są jednoczesne, jeśli żadna z nich nie została wykonana przed drugą (zob. punkt „Relacja »zdarzyło się wcześniej« i jednoczesność”). Można ująć to inaczej: dwa zdarzenia są uporządkowane, jeśli są związane przyczynowo (jedno nastąpiło przed drugim, są jednak nieporównywalne, jeżeli są jednoczesne. To oznacza, że przyczynowość wyznacza *uporządkowanie częściowe*, a nie całkowite. Niektóre operacje są uporządkowane względem innych, pozostałe nie.

Dlatego, zgodnie z tą definicją, w liniowej bazie danych nie występują operacje jednoczesne. Konieczna jest jedna oś czasu z całkowitym uporządkowaniem na niej wszystkich operacji. Część żądań może oczekiwać na obsłużenie, jednak baza danych gwarantuje, że każde żądanie jest obsługiwane atomowo w określonym momencie, z użyciem jednej kopii danych, na jednej osi czasowej i bez jednoczesności.

Jednoczesność oznaczałaby, że oś czasu się rozgałęzia i ponownie łączy. W takim scenariuszu operacje z różnych gałęzi są nieporównywalne (jednoczesne). Zetknąłeś się z tym zjawiskiem w rozdziale 5. Na przykład na rysunku 5.14 jest widoczne nie tyle proste uporządkowanie całkowite, ile bezładny zestaw jednocześnie wykonywanych różnych operacji. Strzałki na rysunku oznaczają związki przyczynowe — uporządkowanie częściowe operacji.

Jeśli znasz rozproszone systemy kontroli wersji (takie jak Git), możesz zauważyć, że historia wersji bardzo przypomina w nich graf związków przyczynowych. Często kolejne zmiany są przesyłane jedna po drugiej w prostej kolejności, jednak czasem pojawiają się odgańlenia (gdy kilka osób jednocześnie pracuje nad projektem), a scalane wersje są tworzone w wyniku łączenia jednocześnie utworzonych przesyłanych zmian.

## Liniowość zapewnia silniejsze gwarancje niż spójność przyczynowa

Jaki jest więc związek między uporządkowaniem przyczynowym a liniowością? Odpowiedź jest taka, że z liniowości *wynika* przyczynowość. Każdy system liniowy poprawnie zachowuje przyczynowość [7]. Zwłaszcza gdy w systemie istnieje wiele kanałów komunikacji (np. kolejka komunikatów i usługa składowania danych w plikach na rysunku 9.5), liniowość gwarantuje, że przyczynowość jest automatycznie zapewniana bez konieczności wykonywania przez system specjalnych operacji (takich jak przekazywanie znaczników czasu między różnymi komponentami).

To, że liniowość gwarantuje przyczynowość, sprawia, iż systemy liniowe są proste do zrozumienia i atrakcyjne. Jednak, co opisano w punkcie „Koszty liniowości”, zapewnianie liniowości systemu może zaszkodzić jego wydajności i dostępności — zwłaszcza w sytuacji, gdy opóźnienia sieciowe są wysokie (czyli np. w systemach rozproszonych geograficznie). Z tego powodu w niektórych rozproszonych systemach danych zrezygnowano w liniowości, co pozwala uzyskać wyższą wydajność, ale utrudnia korzystanie z systemu.

Dobra wiadomość to ta, że możliwe jest rozwiązanie pośrednie. Liniowość nie jest jedynym sposobem zapewniania przyczynowości. Istnieją też inne metody. System może być spójny przyczynowo bez spadku wydajności związanego z zapewnianiem liniowości (przede wszystkim nie ma tu zastosowania twierdzenie CAP). Spójność przyczynowa jest najsilniejszym modelem spójności, który nie spowalnia pracy z powodu opóźnień sieciowych i pozostaje dostępny mimo awarii sieci [2, 42].

W wielu sytuacjach systemy na pozór wymagające liniowości tak naprawdę potrzebują tylko spójności przyczynowej, którą można zaimplementować wydajniej. Na podstawie tej obserwacji badacze analizują nowe rodzaje baz zapewniających przyczynowość oraz oferujących wydajność i dostępność na poziomie podobnym jak systemy ostatecznie spójne [49, 50, 51].

Ponieważ są to stosunkowo nowe badania, niewiele odkryć z tego obszaru trafiło na razie do systemów produkcyjnych. Pozostały też do rozwiązania różne trudności [52, 53]. Jest to jednak obiecujący kierunek rozwoju przyszłych systemów.

## Ujmowanie związków przyczynowych

Nie będziemy omawiać tu szczegółowo tego, jak systemy nieliniowe mogą zachowywać spójność przyczynową. Opisane zostały tylko pokrótce niektóre najważniejsze pomysły.

Aby zachować przyczynowość, trzeba wiedzieć, która operacja *nastąpiła przed* daną inną operacją. Jest to uporządkowanie częściowe. Operacje jednoczesne mogą być wykonywane w dowolnej kolejności, jeśli jednak jedna operacja wystąpiła przed inną, muszą one zostać przetworzone w tym samym porządku w każdej replice. Dlatego gdy replika przetwarza operację, musi zagwarantować, że wszystkie przyczynowo wcześniejsze operacje (które nastąpiły wcześniej) zostały już przetworzone. Jeśli brakuje którejś z wcześniejszych operacji, późniejsza musi czekać do czasu przetworzenia tej wcześniejszej.

Do ustalania związków przyczynowych potrzebny jest sposób opisu „wiedzy” węzła w systemie. Jeśli węzeł widział już wartość X, gdy zażądał zapisu Y, to X i Y mogą być przyczynowo powiązane. W takich analizach używane są pytania, jakich można się spodziewać w śledztwie dotyczącym oszustw — czy CEO *wiedział* o X w momencie, gdy podjął decyzję Y?

Techniki ustalania, która operacja wystąpiła przed inną operacją, są podobne do tych opisanych w punkcie „Wykrywanie jednoczesnych zapisów”. Omówiono tam przyczynowość w bazach bez lidera, gdzie trzeba wykrywać jednoczesne zapisy tego samego klucza, aby zapobiec utracie aktualizacji. Spójność przyczynowa wymaga pójsia o krok dalej — trzeba śledzić związki przyczynowe na poziomie całej bazy (a nie tylko jednego klucza). W tym celu można uogólnić wektory wersji [54].

Aby ustalić uporządkowanie przyczynowe, baza musi wiedzieć, która wersja danych została wczytana przez aplikację. To dlatego na rysunku 5.13 w momencie zapisu do bazy przesyłany jest numer wersji wcześniejszej operacji. Podobny pomysł stosuje się do wykrywania konfliktów w algorytmie SSI, co opisano w punkcie „Algorytm SSI”. Gdy transakcja chce zatwierdzić operację, baza sprawdza, czy wersja danych wczytanych przez tę transakcję wciąż jest aktualna. W tym celu baza śledzi, które dane zostały wczytane przez poszczególne transakcje.

## Uporządkowanie według numerów porządkowych

Choć przyczynowość to ważne zagadnienie teoretyczne, śledzenie wszystkich związków przyczynowych może się stać niepraktyczne. W wielu aplikacjach klienty przed zapisaniem czegokolwiek wczytują dużo danych. Nie jest wtedy jasne, czy zapis jest przyczynowo zależny od wszystkich wcześniejszych odczytów, czy tylko od niektórych z nich. Bezpośrednie śledzenie wszystkich wczytanych danych byłoby bardzo kosztowne.

Istnieje jednak lepsze rozwiązanie. Do porządkowania zdarzeń można się posłużyć *numerami porządkowymi* lub *znacznikami czasu*. Znacznik czasu nie może pochodzić z zegara czasu rzeczywistego (czyli zegara fizycznego; powoduje on liczne problemy opisane w punkcie „Zawodne zegary”). Zamiast tego można go uzyskać z *zegara logicznego*, czyli z algorytmu generującego sekwencję liczb identyfikujących operacje. Zwykle używa się do tego liczników zwiększanych dla każdej operacji.

Takie numery porządkowe lub znaczniki czasu są związane (zajmują tylko kilka bajtów) i mogą wyznaczać *uporządkowanie całkowite*. Oznacza to, że każda operacja ma unikatowy numer porządkowy i zawsze można porównać dwa takie numery, aby ustalić, który z nich jest większy (czyli która operacja nastąpiła później).

Przed wszystkim można tworzyć numery porządkowe zgodnie z uporządkowaniem całkowitym *spójnym z przyczynowością*<sup>7</sup>. Stanowi to obietnicę, że jeśli operacja A przyczynowo wystąpiła przed B, to A nastąpiła przed B w uporządkowaniu całkowitym (A ma niższy numer porządkowy niż B). Operacje jednoczesne mogą być uporządkowane w dowolny sposób. Takie uporządkowanie całkowite ujmuje wszystkie informacje o przyczynowości, a jednocześnie wymusza dokładniejsze uporządkowanie niż to, które jest ściśle wymagane w związku z przyczynowością.

W bazie z replikacją z jednym liderem (zob. punkt „Liderzy i obserwatorzy”) dziennik replikacji definiuje uporządkowanie całkowite operacji zapisu spójne z przyczynowością. Lider może zwiększać licznik w przypadku kolejnych operacji, przypisując w ten sposób monotonicznie rosnące numery

---

<sup>7</sup> Łatwo można uzyskać porządkowanie całkowite, które jest *niespójne* z przyczynowością, nie jest to jednak przydatne. Możesz np. generować dla każdej operacji losowy identyfikator UUID i porównywać te identyfikatory leksykograficznie, aby zdefiniować uporządkowanie całkowite operacji. Jest to poprawne uporządkowanie całkowite, jednak losowe identyfikatory UUID nie informują o tym, która operacja wystąpiła jako pierwsza lub czy operacje były jednoczesne.

porządkowe każdej operacji z dziennika replikacji. Jeśli obserwator wprowadza zapisy w kolejności ich występowania w dzienniku replikacji, stan obserwatora jest zawsze spójny przyczynowo (nawet gdy obserwator jest opóźniony względem lidera).

### Nieprzyczynowe generatory numerów porządkowych

Jeśli nie istnieje jeden lider (np. z powodu stosowania bazy z wieloma liderami lub bez lidera albo z przyczyny podziału bazy na partycje), sposób generowania numerów porządkowych operacji jest mniej oczywisty. W praktyce stosuje się różne metody:

- Każdy węzeł może generować własny, niezależny zbiór numerów porządkowych. Na przykład jeśli istnieją dwa węzły, jeden może generować tylko numery nieparzyste, a drugi same numery parzyste. Część bitów w binarnej reprezentacji numeru porządkowego można zarezerwować na unikatowy identyfikator węzła, co gwarantuje, że dwa różne węzły nigdy nie wygenerują tego samego numeru porządkowego.
- Możesz dodawać do każdej operacji znacznik czasu na podstawie zegara czasu rzeczywistego (zegara fizycznego) [55]. Takie znaczniki czasu nie są sekwencyjne, jeśli jednak są wystarczająco dokładne, mogą wystarczyć do zapewnienia uporządkowania całkowitego operacji. Ten fakt jest wykorzystywany w metodzie rozwiązywania konfliktów „ostatni zapis wygrywa” (zob. punkt „Używanie znaczników czasu do porządkowania zdarzeń”).
- Możesz wstępnie przydzielać bloki numerów porządkowych. Na przykład węzeł A może otrzymać blok numerów od 1 do 1000, a węzeł B od 1001 do 2000. Następnie każdy węzeł może niezależnie przypisywać numery porządkowe ze swojego bloku i uzyskiwać nowy blok, gdy dostępne numery zaczną się kończyć.

Każde z tych trzech rozwiązań jest wydajniejsze i bardziej skalowalne niż wykonywanie wszystkich operacji w jednym liderze zwiększającym licznik. W tych podejściach generowane są unikatowe, rosnące numery porządkowe dla każdej operacji. Jednak we wszystkich tych technikach występuje problem: generowane numery porządkowe *nie są spójne z przyczynowością*.

Problem przyczynowości pojawia się, ponieważ generatory liczb porządkowych nie uwzględniają poprawnie kolejności operacji z różnych węzłów:

- Każdy węzeł może przetwarzać inną liczbę operacji na sekundę. Dlatego jeśli jeden węzeł generuje liczby parzyste, a drugi liczby nieparzyste, licznik dla liczb parzystych może być opóźniony względem licznika liczb nieparzystych (lub na odwrót). Jeżeli jedna operacja ma numer parzysty, a druga nieparzysty, nie można określić, która z nich przyczynowo została wykonana jako pierwsza.
- Znaczniki czasu z zegarów fizycznych są narażone na odchylenia zegarów, przez co mogą być niespójne z przyczynowością. Na przykład na rysunku 8.3 pokazany jest scenariusz, w którym operacja wykonywana przyczynowo później otrzymała niższy znacznik czasu<sup>8</sup>.

---

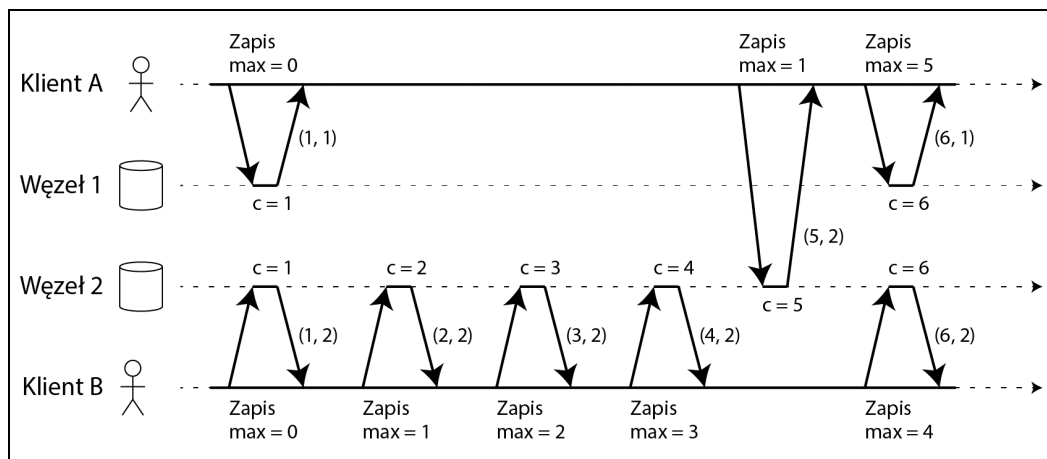
<sup>8</sup> Można zapewnić spójność znaczników czasu z fizycznych zegarów z przyczynowością. W punkcie „Synchronizowane zegary używane do tworzenia globalnych snapshotów” opisano bazę Spanner firmy Google, która to baza szacuje oczekiwane odchylenie zegara i przed zatwierdzeniem zapisu odczekuje czas, o jaki może się mylić. Ta metoda gwarantuje, że przyczynowo późniejsza transakcja otrzyma wyższy znacznik czasu. Jednak większość zegarów nie potrafi udostępnić potrzebnego poziomu możliwego błędu.

- Gdy stosowane są bloki numerów, jedna operacja może otrzymać numer porządkowy z przedziału od 1001 do 2000, a przyczynowo późniejsza operacja z zakresu od 1 do 1000. Także tu numery porządkowe są niespójne z przyczynowością.

## Znaczniki czasu Lamporta

Choć trzy opisane generatory numerów porządkowych są niespójne z przyczynowością, istnieje prosta metoda generowania numerów *spójnych* z przyczynowością. Jej nazwa to *znaczniki czasu Lamporta*. Ta technika została zaproponowana w 1978 r. przez Lesliego Lamporta [56] w jednym z obecnie najczęściej cytowanych artykułów z dziedziny systemów rozproszonych.

Wykorzystanie znaczników czasu Lamporta jest pokazane na rysunku 9.8. Każdy węzeł ma unikatowy identyfikator. Ponadto każdy węzeł przechowuje licznik przetworzonych operacji. Znacznik czasu to para (*licznik*, *identyfikator węzła*). Dwa węzły mogą czasem mieć tę samą wartość licznika, jednak dodanie identyfikatora węzła w każdym znaczniku czasu zapewnia unikatowość tych znaczników.



Rysunek 9.8. Znaczniki czasu zapewniają uporządkowanie całkowite spójne z przyczynowością

Znaczniki czasu Lamporta nie są powiązane z zegarem czasu rzeczywistego, jednak zapewniają uporządkowanie całkowite. Jeśli istnieją dwa znaczniki czasu, ten z większą wartością licznika jest większy. Gdy wartości licznika są takie same, znacznik z wyższym identyfikatorem węzła jest większy.

Do tej pory ten opis jest w zasadzie identyczny z przedstawionymi w poprzednim punkcie licznikami parzystymi i nieparzystymi. Najważniejszą cechą znaczników czasu Lamporta zapewniającą ich spójność z przyczynowością jest to, że wszystkie węzły i klienci przechowują *maksymalną* widzianą do danej chwili wartość licznika i umieszczają to maksimum w każdym żądaniu. Gdy węzeł otrzyma żądanie lub odpowiedź z maksymalną wartością licznika wyższą niż wartość jego licznika, natychmiast przestawia własny licznik na wartość maksymalną.

Jest to widoczne na rysunku 9.8, gdzie klient A otrzymuje od węzła 2 wartość licznika 5, a następnie przesyła tę wartość do węzła 1. Wtedy licznik węzła 1 ma wartość 1, jednak zostaje ona natychmiast podniesiona do 5, dlatego następna operacja zwiększa wartość licznika do 6.

Dopóki maksymalna wartość licznika jest przekazywana razem z każdą operacją, ten system gwarantuje, że uporządkowanie oparte na znacznikach czasu Lamporta jest spójne z przyczynowością, ponieważ każda zależność przyczynowa skutkuje zwiększeniem znacznika czasu.

Znaczniki czasu Lamporta są czasem mylone z wektorami wersji przedstawionymi w punkcie „Wykrywanie jednoczesnych zapisów”. Choć między tymi technikami występują pewne podobieństwa, te metody są przeznaczone do czegoś innego. Wektory wersji pozwalają stwierdzić, czy dwie operacje są jednoczesne, czy jedna jest przyczynowo zależna od drugiej. Z kolei znaczniki czasu Lamporta zawsze wymuszają uporządkowanie całkowite. Na podstawie uporządkowania czasowego wyznaczonego przez znaczniki czasu Lamporta nie da się stwierdzić, czy dwie operacje są jednoczesne, czy zależne przyczynowo. Przewagą znaczników czasu Lamporta nad wektorami wersji jest ich większa zwężłość.

### Porządkowanie za pomocą znaczników czasu to za mało

Choć znaczniki czasu Lamporta definiują uporządkowanie całkowite operacji spójne z przyczynowością, nie wystarczają do rozwiązania wielu standardowych problemów z obszaru systemów rozproszonych.

Wyobraź sobie np. system, który musi gwarantować, że nazwa użytkownika stanowi unikatowy identyfikator konta. Jeśli dwóch użytkowników jednocześnie próbuje utworzyć konto o tej samej nazwie, jednemu z nich powinno się to udać, a drugiemu nie. O tym problemie wspomniano wcześniej w punkcie „Lider i blokady”.

Na pozór wydaje się, że uporządkowanie całkowite operacji (np. za pomocą znaczników czasu Lamporta) wystarcza do rozwiązania tego problemu. Jeśli tworzone są dwa konta o tej samej nazwie, należy uznać za zwycięzcę to o niższym znaczniku czasu (dla którego wcześniej zarezerwowano nazwę). Próba użytkownika z wyższym znacznikiem czasu powinna zakończyć się niepowodzeniem. Ponieważ dla znaczników czasu zawsze istnieje uporządkowanie całkowite, takie porównanie jest zawsze prawidłowe.

To podejście pozwala ustalić zwycięzcę po fakcie. Po zarejestrowaniu wszystkich operacji tworzenia nazwy użytkownika w systemie można porównać ich znaczniki czasu. Jednak to rozwiązanie nie wystarcza, gdy węzeł właśnie otrzymał od użytkownika żądanie utworzenia nazwy i musi zdecydować *w tym momencie*, czy żądanie powinno zakończyć się powodzeniem, czy porażką. W danej chwili węzeł nie wie, czy inny węzeł przetwarza tworzenie konta o tej samej nazwie i jaki znacznik czasu ten inny węzeł przypisał wykonywanej operacji.

Aby mieć pewność, że żaden inny węzeł nie tworzy w tym samym momencie konta o tej samej nazwie i z niższym znacznikiem czasu, trzeba sprawdzić działania wszystkich pozostałych węzłów [56]. Gdy jeden z innych węzłów uległ awarii lub jest niedostępny z powodu problemów z siecią, system się zatrzymuje. Nie jest wtedy odpornym na błędy systemem, na jakim nam zależy.

Problem polega na tym, że uporządkowanie całkowite operacji wyłania się dopiero po zarejestrowaniu ich wszystkich. Jeśli inny węzeł uruchomił jakieś operacje, ale jeszcze nie wie, na czym polegają, nie da się uzyskać uporządkowania całkowitego. Możliwe, że na różnych pozycjach w uporządkowaniu całkowitym trzeba będzie wstawić nieznanne operacje z innego węzła.

W podsumowaniu: aby zaimplementować coś w rodzaju więzów unikatowości dla nazw użytkownika, nie wystarczy uporządkowanie całkowite operacji. Trzeba też wiedzieć, kiedy to uporządkowanie jest finalizowane. Jeśli operacja tworzy nazwę użytkownika i z pewnością wiadomo, że żaden inny węzeł nie mógł zgodnie z uporządkowaniem całkowitym zażądać tej nazwy przed daną operacją, można bezpiecznie uznać operację za udaną. Informacja o tym, kiedy uporządkowanie całkowite z danej jednostki jest finalizowane, jest ujęta w *rozgłaszaniu z uporządkowaniem całkowitym*.

## Rozgłaszanie z uporządkowaniem całkowitym

Jeśli program działa w procesorze jednordzeniowym, łatwo jest zdefiniować uporządkowanie całkowite operacji. Jest to kolejność ich wykonywania w tym procesorze. Jednak w systemie rozproszonym uzgodnienie przez wszystkie węzły tego samego uporządkowania całkowitego operacji jest trudne. W ostatnim punkcie opisano porządkowanie na podstawie znaczników czasu lub numerów porządkowych, jednak okazało się, że te techniki nie są równie skuteczne jak replikacja z jednym liderem (zastosowanie uporządkowania na podstawie znaczników czasu do implementacji więzów unikatowości nie zapewnia odporności na błędy).

Opisano, że w replikacji z jednym liderem uporządkowanie całkowite operacji uzyskuje się przez wybór jednego węzła jako lidera i porządkowanie wszystkich operacji w jednym rdzeniu procesora tego lidera. Trudność polega tu na skalowaniu systemu, jeśli obciążenie jest tak wysokie, że jeden lider sobie z nim nie radzi. Problem stanowi też obsługa przełączania awaryjnego po ustercie lidera (zob. punkt „Radzenie sobie z przestojami węzłów”). W literaturze poświęconej systemom rozproszonym ten problem jest nazywany *rozgłaszaniem z uporządkowaniem całkowitym* (ang. *total order broadcast*) lub *rozgłaszaniem atomowym* (ang. *atomic broadcast*) [25, 57, 58]<sup>9</sup>.



### Zakres gwarancji uporządkowania

W bazach podzielonych na partycje, gdzie w każdej partycji działa jeden lider, często uporządkowanie jest określane tylko dla partycji. Oznacza to, że baza nie może zapewnić gwarancji spójności (np. spójnych snapshotów, referencji w postaci kluczy zewnętrznych) na poziomie wszystkich partycji. Uporządkowanie całkowite na poziomie wszystkich partycji jest możliwe, jednak wymaga dodatkowej koordynacji [59].

Rozgłaszanie uporządkowania całkowitego jest zazwyczaj opisywane jako protokół wymiany komunikatów między węzłami. Nieformalnie podejście to wymaga, by zawsze spełnione były dwie cechy związane z bezpieczeństwem:

#### *Niezawodne dostarczanie*

Żadne komunikaty nie zostają utracone. Jeśli komunikat dociera do jednego węzła, zostaje dostarczony do wszystkich.

#### *Dostarczanie zgodnie z uporządkowaniem całkowitym*

Komunikaty są dostarczane do każdego węzła w tej samej kolejności.

<sup>9</sup> Tradycyjne określenie to *rozgłaszanie atomowe*, jest ono jednak bardzo mylące, ponieważ jest niezgodne z innymi zastosowaniami słowa *atomowy*. Nie ma ono nic wspólnego z atomowością w transakcjach ACID i jest tylko pośrednio powiązane z operacjami atomowymi (w sensie programowania wielowątkowego) oraz rejestrami atomowymi (bazami liniowymi). Jeszcze inny synonim to *rozsyłanie z uporządkowaniem całkowitym* (ang. *total order multicast*).

Prawidłowy algorytm rozgłaszania z uporządkowaniem całkowitym musi gwarantować, że niezawodność i kolejność są zapewniane zawsze — nawet gdy węzeł lub sieć działają nieprawidłowo. Oczywiście w czasie, gdy sieć nie działa, komunikaty nie są dostarczane. Jednak algorytm ponawia próby, dlatego komunikaty docierają do celu po ostatecznym naprawieniu sieci (i muszą wtedy zostać dostarczone we właściwej kolejności).

## Stosowanie rozgłaszania z uporządkowaniem całkowitym

Usługi zapewniania konsensusu, np. ZooKeeper i etcd, stosują rozgłaszanie z uporządkowaniem całkowitym. Ten fakt wskazuje na to, że występuje ścisłe powiązanie między takim rozgłaszaniem a osiągnięciem konsensusu, co jest omówione w dalszej części rozdziału.

Rozgłaszanie z uporządkowaniem całkowitym to właśnie to, co jest potrzebne do replikacji baz. Jeśli każdy komunikat reprezentuje zapis w bazie, a każda replika przetwarza te same zapisy w tej samej kolejności, repliki pozostaną spójne ze sobą (jeśli pominąć tymczasowe opóźnienie replikacji). Ta zasada to *replikacja maszyny stanowej* [60]. Wrócimy do niej w rozdziale 11.

Rozgłaszanie z uporządkowaniem całkowitym można też zastosować do implementowania transakcji sekwencyjnych. W punkcie „Rzeczywiste sekwencyjne wykonywanie transakcji” opisano, że jeśli każdy komunikat reprezentuje deterministyczną transakcję wykonywaną jako procedura składowana, a każdy węzeł przetwarza te komunikaty w tej samej kolejności, partycje i repliki bazy danych pozostają spójne ze sobą [61].

Ważnym aspektem rozgłaszania z uporządkowaniem całkowitym jest to, że kolejność jest ustalona już w momencie dostarczenia komunikatów. Węzeł nie może po czasie wstawić komunikatu na wcześniejszą pozycję, jeśli późniejsze komunikaty już zostały dostarczone. To sprawia, że takie rozgłaszanie jest silniejsze od porządkowania z użyciem znaczników czasu.

Rozgłaszanie z uporządkowaniem całkowitym można też traktować jak sposób tworzenia *dziennika* (np. dziennika replikacji, dziennika transakcji lub dziennika zapisu z wyprzedzeniem). Dostarczenie komunikatu przypomina dodanie danych do dziennika. Ponieważ wszystkie węzły muszą dostarczyć te same komunikaty w tej samej kolejności, wszystkie węzły mogą wczytać dane z dziennika i zobaczyć tę samą sekwencję komunikatów.

Rozgłaszanie z uporządkowaniem całkowitym jest przydatne także do implementowania usługi obsługi blokad, która zapewnia tokeny odgradzające (zob. punkt „Tokeny odgradzające”). Każde żądanie blokady jest dołączane do dziennika jako komunikat, a wszystkie komunikaty otrzymują numery porządkowe zgodnie z kolejnością dodawania ich do dziennika. Numery porządkowe mogą następnie posłużyć za tokeny odgradzające, ponieważ są monotonicznie rosnące. W ZooKeeperze te numery porządkowe to `zxid` [15].

## Implementowanie bazy liniowej za pomocą rozgłaszania z uporządkowaniem całkowitym

Na rysunku 9.4 pokazano, że w systemie liniowym występuje uporządkowanie całkowite operacji. Czy to oznacza, że liniowość jest tym samym co rozgłaszanie z uporządkowaniem całkowitym? Nie do końca, jednak oba te zagadnienia są ze sobą ściśle powiązane<sup>10</sup>.

<sup>10</sup> W sensie formalnym tworzenie liniowego rejestru z zapisem i odczytem to „łatwiejszy” problem. Rozgłaszanie z uporządkowaniem całkowitym to odpowiednik problemu osiągnięcia konsensusu [67], dla którego nie istnieje determini-



Rozgłaszanie z uporządkowaniem całkowitym jest asynchroniczne. Gwarantuje, że komunikaty zostaną niezawodnie dostarczone w ustalonej kolejności. Nie ma jednak gwarancji, *kiedy* komunikat zostanie dostarczony (dlatego jeden odbiorca może być opóźniony względem innych). Z kolei liniowość to gwarancja nowości. Gwarantuje, że odczyt zwróci najnowszą zapisaną wartość.

Jeśli jednak stosujesz rozgłaszanie z uporządkowaniem całkowitym, możesz na tej podstawie zbudować bazę liniową. Możesz np. zagwarantować, że nazwy użytkowników unikatowo identyfikują konta.

Wyobraź sobie, że dla każdej nazwy stosuje się rejestr liniowy z atomową operacją „porównaj i ustaw”. Każdy rejestr początkowo ma wartość `null` (informującą, że nazwa nie jest zajęta). Gdy użytkownik chce wykorzystać daną nazwę, należy wykonać operację „porównaj i ustaw” dla rejestru powiązanego z tą nazwą i — jeśli wcześniejsza wartość rejestru to `null` — ustawić go na identyfikator konta. Jeśli kilku użytkowników próbuje jednocześnie zająć tę samą nazwę, tylko jedna operacja „porównaj i ustaw” zakończy się powodzeniem, ponieważ w pozostałych wartość będzie różna od `null` (z powodu liniowości).

Tego rodzaju liniową operację „porównaj i ustaw” możesz zaimplementować w następujący sposób, stosując rozgłaszanie z uporządkowaniem całkowitym jak dziennik tylko z funkcją dołączania danych [62, 63]:

1. Dołącz komunikat do dziennika, wstępnie określając nazwę użytkownika, jaką chcesz zająć.
2. Wczytaj dziennik i odczekaj do czasu, aż dołączony komunikat zostanie Ci dostarczony<sup>11</sup>.
3. Poszukaj innych komunikatów z żądaniem zajęcia danej nazwy. Jeśli pierwszy komunikat dotyczący tej nazwy należy do Ciebie, udało się — możesz zatwierdzić żądanie nazwy (prawdopodobnie przez dołączenie do dziennika następnego komunikatu) i potwierdzić to klientowi. Jeżeli pierwszy komunikat związany z nazwą pochodzi od innego użytkownika, własną operację należy anulować.

Ponieważ wpisy z dziennika są dostarczane do wszystkich węzłów w tej samej kolejności, to jeśli występuje kilka jednoczesnych zapisów, wszystkie węzły zgadzają się co do tego, który z nich pojawił się pierwszy. Wybór pierwszego ze sprzecznych zapisów jako zwycięzcy i anulowanie późniejszych gwarantuje, że wszystkie węzły zgadzają się w kwestii tego, czy zapis został zatwierdzony, czy anulowany. Podobne podejście można zastosować do implementowania sekwencyjnych transakcji na wielu obiektach z użyciem dziennika [62].

Choć opisany proces gwarantuje liniowe zapisy, nie gwarantuje liniowych odczytów. Jeśli wczytujesz dane z bazy, która jest asynchronicznie aktualizowana na podstawie dziennika, dane mogą być nieaktualne. W precyzyjnym ujęciu opisany tu proces zapewnia *spójność sekwencyjną* [47, 64], którą można też nazwać *spójnością czasową* (ang. *timeline consistency*) [65, 66]; jest to gwarancja nieco słabsza od liniowości. Istnieje kilka sposobów na zapewnienie liniowości odczytów:

---

styczne rozwiązywanie w asynchronicznym modelu awaria – przerwanie pracy [68]. Natomiast liniowy rejestr z zapisem i odczytem *można* zaimplementować w tym modelu. Jednak dodanie obsługi operacji atomowych na rejestrze (takich jak „porównaj i ustaw” lub „zwiększ i pobierz”) sprawia, że ten problem staje się odpowiednikiem osiągnięcia konsensusu [28]. Dlatego problemy osiągnięcia konsensusu i rejestru liniowego są ze sobą ściśle powiązane.

<sup>11</sup> Jeśli pominiesz oczekiwanie i będziesz potwierdzał zapis natychmiast po jego umieszczeniu w kolejce, uzyskasz coś podobnego do modelu spójności pamięci z wielordzeniowych procesorów x86 [43]. Ten model nie jest ani liniowy, ani spójny sekwencyjnie.

- Możesz zapewnić sekwencyjność odczytów, dodając komunikat, wczytując dziennik i przeprowadzając odczyt dopiero po otrzymaniu tego komunikatu. Pozycja komunikatu w dzienniku definiuje więc moment odczytu. Podobnie działają odczyty z użyciem kworum w bazie etcd [16].
- Jeśli dziennik umożliwia pobranie pozycji ostatniego komunikatu w liniowy sposób, możesz sprawdzić tę pozycję, poczekać na dostarczenie wszystkich wpisów do tej pozycji, a następnie przeprowadzić odczyt. Na tej zasadzie działa operacja `sync()` w ZooKeeperze [15].
- Możesz wczytywać dane z repliki, która jest synchronicznie aktualizowana przy zapisie, dlatego z pewnością jest aktualna. Tę technikę stosuje się w replikacji łańcuchowej [63]. Zob. też ramkę „Badania nad replikacją”.

## Implementowanie rozgłaszania z uporządkowaniem całkowitym za pomocą bazy liniowej

W poprzednim punkcie pokazano, jak uzyskać liniową operację „porównaj i ustaw” za pomocą rozgłaszania z uporządkowaniem całkowitym. Można też odwrócić sytuację i przy założeniu, że dostępna jest baza liniowa, uzyskać rozgłaszanie z uporządkowaniem całkowitym.

Najłatwiej jest przyjąć, że dostępny jest rejestr liniowy przechowujący liczbę całkowitą i obsługujący atomową operację „zwiększ i pobierz” [28]. Odpowiednia będzie też atomowa operacja „porównaj i ustaw”.

Algorytm jest prosty — dla każdego komunikatu, jaki chcesz przesłać za pomocą rozgłaszania z uporządkowaniem całkowitym, należy zwiększyć i pobrać liniową liczbę całkowitą, a następnie dołączyć wartość z rejestru jako numer porządkowy do komunikatu. Następnie możesz przesłać komunikaty do wszystkich węzłów (ponawiając próby dostarczenia wszystkich utraconych komunikatów), a odbiorcy uporządkują komunikaty w odpowiedniej kolejności zgodnie z numerami porządkowymi.

Zauważ, że liczby uzyskane dzięki zwiększaniu liniowego rejestru tworzą (w odróżnieniu od znaczników czasu Lamporta) sekwencję bez luk. Dlatego jeśli węzeł dostarczył komunikat nr 4 i otrzymał komunikat nr 6, wie, że przed dostarczeniem komunikatu nr 6 musi poczekać na komunikat nr 5. Gdy stosujesz znaczniki czasu Lamporta, jest inaczej. Stanowi to główną różnicę między rozgłaszaniem z uporządkowaniem całkowitym a porządkowaniem na podstawie znaczników czasu.

Jak trudno uzyskać liniową liczbę całkowitą z atomową operacją „zwiększ i pobierz”? Jak zwykle jest tak, że gdyby nie występowały awarie, zadanie byłoby łatwe. Wystarczyłoby przechowywać wartość w zmiennej w jednym węźle. Problem polega na obsłudze sytuacji, gdy połączenia sieciowe z tym węzłem zostają przerwane i gdy trzeba odzyskać wartość po awarii tego węzła [59]. Jeśli się dobrze zastanowisz nad liniowymi generatorami liczb porządkowych, z pewnością dojdiesz do algorytmu osiągnięcia konsensusu.

To nie przypadek. Można udowodnić, że liniowy rejestr z operacją „porównaj i ustaw” (lub „zwiększ i pobierz”) oraz rozgłaszanie z uporządkowaniem całkowitym są odpowiednikami *osiągania konsensusu* [28, 67]. Oznacza to, że jeśli rozwiążesz jeden z tych problemów, po transformacji otrzymasz rozwiązanie pozostałych. To ważne i zaskakujące spostrzeżenie.

Pora wreszcie bezpośrednio zmierzyć się z problemem osiągnięcia konsensusu. Tego dotyczy reszta rozdziału.

# Transakcje rozproszone i konsensus

Osiąganie konsensusu to jeden z najważniejszych i najbardziej podstawowych problemów w obszarze przetwarzania rozproszonego. Pozornie jest to proste zadanie. W nieformalnym ujęciu celem jest *zapewnienie zgody kilku węzłów w jakiejś kwestii*. Może Ci się wydawać, że nie jest to trudne. Niestety, wiele nieprawidłowych systemów zostało zbudowanych z błędnym przekonaniem, że jest to łatwy problem.

Choć konsensus jest bardzo ważny, poświęcony mu punkt pojawia się daleko w książce, ponieważ jest to złożony temat, a do zrozumienia związanych z nim subtelności niezbędna jest wstępna wiedza. Nawet w społeczności naukowców zrozumienie konsensusu stopniowo krystalizowało się przez dziesięciolecia, kiedy to pojawiały się liczne błędy. Teraz, po omówieniu replikacji (rozdział 5.), transakcji (rozdział 7.), modeli systemów (rozdział 8.), liniowości i rozgłaszania z uporządkowaniem całkowitym (ten rozdział), jesteśmy wreszcie gotowi zająć się problemem osiągnięcia konsensusu.

Zgoda między węzłami jest ważna w wielu sytuacjach. Oto przykłady:

## *Wybór lidera*

W bazie z replikacją z jednym liderem wszystkie węzły muszą się zgadzać co do tego, który z nich jest liderem. Pozycja lidera może zostać podważona, jeśli niektóre węzły z powodu błędu sieci nie komunikują się z innymi. Wtedy konsensus jest ważny, aby uniknąć kłopotliwego przełączania awaryjnego, które mogłoby prowadzić do „rozszerzenia mózgu”, kiedy to dwa węzły uważają, że są liderem (zob. punkt „Radzenie sobie z awariami węzłów”). Jeśli działa dwóch liderów, obaj akceptują zapisy i mogą przechowywać różne dane, co skutkuje niespójnością i utratą danych.

## *Atomowe zatwierdzanie*

W bazie z obsługą transakcji, która obejmuje kilka węzłów lub partycji, występuje ten problem, że transakcja może się udać w jednych węzłach, ale nie powieść w innych. Jeśli chcesz zachować atomowość transakcji (w sensie modelu ACID; zob. punkt „Atomowość”), musisz zapewnić zgodność wszystkich węzłów co do skutków transakcji. Albo wszystkie węzły anulują i wycofują transakcję (jeśli coś się nie powiodło), albo wszystkie ją zatwierdzają (jeżeli nie wystąpiły problemy). Ten rodzaj osiągnięcia konsensusu jest nazywany problemem *atomowego zatwierdzania*<sup>12</sup>.

W tym punkcie najpierw szczegółowo omówiono problem atomowego zatwierdzania. Analizowany jest zwłaszcza algorytm *dwuetapowego zatwierdzania*, który stanowi najbardziej popularne rozwiązanie problemu atomowego zatwierdzania i jest stosowany w różnych bazach, systemach przesyłania komunikatów i serwerach aplikacji. Okazuje się, że algorytm dwuetapowego zatwierdzania to algorytm osiągnięcia konsensusu, choć niezbyt dobry [70, 71].

<sup>12</sup> Atomowe zatwierdzanie w formalnej postaci różni się nieco od osiągnięcia konsensusu. Atomową transakcję można zatwierdzić tylko wtedy, jeśli wszyscy uczestnicy zgłoszą w ten sposób. Jeśli którykolwiek uczestnik zażąda anulowania, transakcję trzeba wycofać. W ramach konsensusu można zdecydować się na *dowolną* wartość zaproponowaną przez jednego z uczestników. Jednak atomowe zatwierdzanie można zredukować do osiągnięcia konsensusu i na odwrót [70, 71]. Atomowe zatwierdzanie *bez blokowania* jest trudniejsze do uzyskania niż konsensus; zob. punkt „Trzyetapowe zatwierdzanie”.

## Niemożliwość osiągnięcia konsensusu

Możliwe, że słyszałeś o dowodzie FLP [68] (nazwa ta pochodzi od autorów: Fischera, Lyncha i Pater-sona). Wynika z niego, że jeśli występuje ryzyko awarii, nie istnieje algorytm, który zawsze potrafi za-  
pewnić konsensus. W systemie rozproszonym trzeba przyjąć, że węzły mogą ulec awarii, dlatego nie-  
zawodne osiąganie konsensusu jest niewykonalne. Jednak w tym miejscu omawiane są algorytmy  
osiągania konsensusu. O co tu chodzi?

Odpowiedź jest taka, że dowód FLP obowiązuje w modelu asynchronicznym (zob. punkt „Model sys-  
temu a rzeczywistość”). Jest to bardzo restrykcyjny model oparty na deterministycznym algorytmie,  
który nie może korzystać z zegarów ani limitów czasu. Jeśli w algorytmie dozwolone są limity czasu  
lub inny sposób wykrywania węzłów podejrzanych o awarię (nawet jeżeli te podejrzenia okazują się  
czasem nieuzasadnione), problem osiągania konsensusu staje się możliwy do rozwiązania [67]. Nawet  
dopuszczenie w algorytmie stosowania liczb losowych wystarcza, by wspomniany dowód przestał  
obowiązywać [69].

Dlatego choć dowód FLP na niemożność osiągnięcia konsensusu ma wielkie znaczenie teoretyczne,  
w praktyce w systemach rozproszonych konsensus zwykle jest osiągany.

Po wyciągnięciu wniosków z algorytmu dwuetapowego zatwierdzania przejdziemy do lepszych algo-  
rytmów osiągania konsensusu, takich jak stosowane są w ZooKeeperze (Zab) i bazie etcd (Raft).

## Atomowe zatwierdzanie i dwuetapowe zatwierdzanie

Z rozdziału 7. dowiedziałeś się, że atomowość transakcji ma zapewniać prostą semantykę kodu  
w sytuacjach, gdy coś się nie powiedzie w trakcie wykonywania kilku zapisów. Skutkiem transakcji  
jest albo jej udane *zatwierdzenie*, kiedy to wszystkie zapisy transakcji są utrwalane, albo jej *anulowa-  
nie*, kiedy to wszystkie zapisy są wycofywane (cofane lub odrzucane).

Atomowość zabezpiecza przed zaśmiecaniem bazy przez nieudane transakcje częściowo gotowy-  
mi wynikami i częściowo zaktualizowanym stanem. Jest to ważne zwłaszcza dla transakcji na  
wielu obiektach (zob. punkt „Operacje na pojedynczych obiektach i na wielu obiektach”) i dla baz  
z indeksami pomocniczymi. Każdy indeks pomocniczy to struktura danych odrębna od głównych  
danych. Dlatego jeśli modyfikujesz jakieś dane, zmiany trzeba wprowadzić także w indeksie pomo-  
cniczym. Atomowość gwarantuje, że indeksy pomocnicze pozostają spójne z głównymi danymi  
(jeśli indeks stanie się niespójny z tymi danymi, będzie nieprzydatny).

### Zatwierdzanie atomowe w jednym węźle i w systemie rozproszonym

W przypadku transakcji wykonywanych w jednym węźle bazy atomowość jest zwykle imple-  
mentowana przez system bazodanowy. Gdy klient żąda zatwierdzenia transakcji od węzła bazy,  
baza utrwała zapisy transakcji (zwykle w dzienniku zapisu z wyprzedzeniem; zob. punkt „Zapew-  
nianie niezawodności b-drzew”), a następnie dołącza rekord zatwierdzający do dziennika na dys-  
ku. Jeśli w tym czasie baza ulegnie awarii, transakcja zostanie odzyskana na podstawie dziennika  
w momencie wznowienia pracy węzła. Jeżeli rekord zatwierdzający został przed awarią z powo-  
dzeniem zapisany na dysku, transakcja jest uznawana za zatwierdzoną. W przeciwnym razie wszyst-  
kie zapisy z transakcji są wycofywane.

Dlatego w jednym węźle zatwierdzanie transakcji zależy od *kolejności*, w jakiej dane są trwale zapisywane na dysku. Najpierw zapisywane są dane, potem rekord zatwierdzający [72]. Najważniejszym momentem określającym, czy transakcja zostanie zatwierdzona, czy anulowana, jest chwila, kiedy dysk kończy zapis rekordu zatwierdzającego. Do tej chwili można anulować transakcję (z powodu awarii), jednak później jest już ona zatwierdzona (nawet jeśli nastąpi awaria bazy). Tak więc jedno urządzenie (kontroler konkretnego dysku podłączonego do danego węzła) sprawia, że zatwierdzanie odbywa się atomowo.

Co się jednak dzieje, jeśli w transakcję zaangażowanych jest wiele węzłów? Możliwe, że wykonujesz transakcję na wielu obiektach w bazie podzielonej na partycje lub w indeksie pomocniczym podzielonym według pojęć (kiedy to wpis z indeksu może się znajdować w innym węźle niż główne dane; zob. punkt „Podział na partycje a indeksy pomocnicze”). Większość baz rozproszonych typu NoSQL nie obsługuje tego rodzaju transakcji rozproszonych. Jednak takie transakcje są obsługiwane przez różne klastrowe systemy relacyjne (zob. punkt „Transakcje rozproszone w praktyce”).

W opisanych sytuacjach nie wystarczy przesłać żądania zatwierdzenia do wszystkich węzłów i niezależnie zatwierdzać transakcję w każdym z nich. W tym podejściu zatwierdzenie może się powieść w niektórych węzłach i nie udać w innych, co narusza gwarancję atomowości:

- Niektóre węzły mogą wykryć naruszenie więzów lub konflikt, co wymaga anulowania, natomiast inne mogą z powodzeniem zatwierdzić transakcję.
- Niektóre żądania zatwierdzenia mogą zostać utracone w sieci, co ostatecznie skutkuje anulowaniem z powodu przekroczenia limitu czasu, natomiast inne docierają do celu.
- Niektóre węzły mogą ulec awarii przed zapisaniem rekordu zatwierdzającego i wycofać transakcję po wznowieniu pracy, natomiast inne z powodzeniem zatwierdzają operację.

Jeśli niektóre węzły zatwierdzają transakcję, a inne ją anulują, węzły stają się niespójne ze sobą (tak jak na rysunku 7.3). Po zatwierdzeniu transakcji w jednym węźle nie da się jej później wycofać, jeśli się okaże, że została anulowana w innym węźle. Dlatego węzeł może zatwierdzić transakcję tylko wtedy, gdy ma pewność, że wszystkie pozostałe powiązane z nią węzły też ją zatwierdzą.

Zatwierdzanie transakcji musi być nieodwracalne. Nie można zmienić zdania i wstecznie anulować już zatwierdzonej transakcji. Wynika to z tego, że dane po zatwierdzeniu stają się widoczne dla innych transakcji, dlatego inne klienty mogą zacząć na nich polegać. Ta reguła stanowi podstawę izolacji na poziomie *odczytu zatwierdzonych danych*, co opisano w punkcie „Odczyt zatwierdzonych danych”. Możliwość anulowania transakcji po jej zatwierdzeniu sprawia, że każda transakcja, która wczytała zatwierdzone dane, jest oparta na danych wstecznie uznanych za nieistniejące (dlatego też wymaga anulowania).

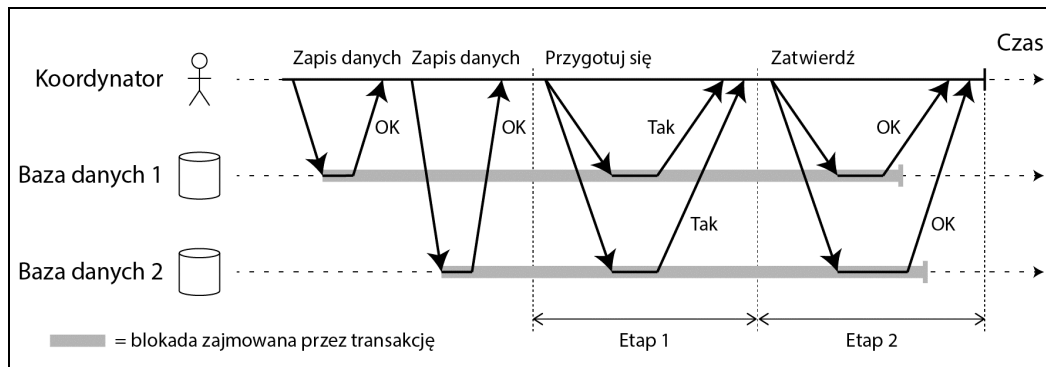
Skutki zatwierdzonej transakcji można później wycofać za pomocą *transakcji kompensującej* [73, 74]. Jednak z perspektywy bazy jest to już odrębna transakcja, a tego typu wymogi zapewniania poprawności na poziomie wielu transakcji są problemem twórcy aplikacji.

## Wprowadzenie zatwierdzania dwuetapowego

Zatwierdzanie dwuetapowe to algorytm atomowego zatwierdzania transakcji na poziomie wielu węzłów. Pozwala to zagwarantować, że albo wszystkie węzły zatwierdzą transakcję, albo wszystkie ją anulują. Jest to klasyczny algorytm dla rozproszonych baz danych [13, 35, 75]. Zatwierdzanie

dwuetapowe jest wewnętrznie stosowane w niektórych bazach, a także udostępniane aplikacjom w postaci *transakcji XA* [76, 77] (obsługiwanych np. w interfejsie Java Transaction API) lub transakcji WS-AtomicTransaction dla usług sieciowych SOAP [78, 79].

Podstawowy przebieg zatwierdzania dwuetapowego jest pokazany na rysunku 9.9. Zamiast jednego żądania zatwierdzenia stosowanego w transakcjach w jednym węźle w zatwierdzaniu dwuetapowym proces zatwierdzania i anulowania jest rozbity na dwie fazy (stąd jego nazwa).



Rysunek 9.9. Udane zatwierdzanie dwuetapowe



#### Nie pomył zatwierdzania dwuetapowego z blokadami dwuetapowymi

Zatwierdzanie dwuetapowe i blokady dwuetapowe (zob. punkt „Blokady dwuetapowe”) to dwie zupełnie różne rzeczy. Zatwierdzanie dwuetapowe służy do atomowego zatwierdzania w bazach danych, natomiast blokady dwuetapowe zapewniają izolację na poziomie sekwencyjności. Aby uniknąć pomyłek, najlepiej jest traktować te mechanizmy jako zupełnie odrębne kwestie i ignorować niefortunne podobieństwo nazw.

W zatwierdzaniu dwuetapowym występuje nowy komponent, który standardowo nie pojawia się w transakcjach w jednym węźle. Jest to *koordynator* (nazywany też *menedżerem transakcji*). Koordynator jest często implementowany jako biblioteka (np. wbudowana w kontener Javy EE) w tym samym procesie aplikacji, który żąda transakcji. Może to być jednak także odrębny proces lub odrębna usługa. Przykładowe koordynatory to Narayana, JOTM, BTM i MSDTC.

Transakcja z zatwierdzaniem dwuetapowym rozpoczyna się standardowo, od odczytu i zapisu danych w wielu węzłach bazy. Węzły bazy są nazywane *uczestnikami* transakcji. Gdy aplikacja jest gotowa do zatwierdzenia transakcji, koordynator rozpoczyna etap 1. Wysyła wtedy do każdego węzła żądanie *przygotuj się* z pytaniem, czy węzły są gotowe do zatwierdzenia transakcji. Następnie koordynator śledzi odpowiedzi od uczestników:

- Jeśli wszyscy uczestnicy odpowiadają „tak”, co oznacza, że są gotowi do zatwierdzenia, koordynator w etapie 2. przesyła żądanie *zatwierdź* i transakcja jest zatwierdzana.
- Jeżeli którykolwiek z uczestników odpowiada „nie”, koordynator w etapie 2. przesyła do wszystkich węzłów żądanie *anuluj*.

Ten proces przypomina tradycyjną ceremonię zawierania małżeństwa w kulturach Zachodu. Duchowny pyta pannę młodą i pana młodego o to, czy chcą zawrzeć małżeństwo, i zwykle otrzymuje od obu stron odpowiedź „tak”. Po uzyskaniu potwierdzenia od obu stron duchowny ogłasza parę

mężem i żoną. Transakcja zostaje zatwierdzona, a to radosne wydarzenie jest ogłaszane wszystkim zgromadzonym. Jeśli któraś ze stron nie powie „tak”, ceremonia zostaje anulowana [73].

## System obietnic

Z tego krótkiego opisu może nie wynikać, dlaczego dwuetapowe zatwierdzanie zapewnia atomowość, choć zatwierdzanie jednoetapowe w kilku węzłach tego nie gwarantuje. Przecież żądania przygotowania się i zatwierdzenia w procesie dwuetapowym też mogą zostać utracone. Co sprawia, że zatwierdzanie dwuetapowe różni się od jednoetapowego?

Aby zrozumieć, dlaczego opisane rozwiązanie działa, trzeba dokładnie przeanalizować proces:

1. Gdy aplikacja chce rozpocząć transakcję rozproszoną, żąda identyfikatora transakcji od koordynatora. Ten identyfikator jest unikatowy na poziomie globalnym.
2. Aplikacja rozpoczyna u każdego uczestnika transakcję typową dla jednego węzła i przypisuje jej globalnie unikatowy identyfikator. Wszystkie odczyty i zapisy odbywają się w jednej z transakcji dla pojedynczych węzłów. Jeśli na tym etapie coś się nie powiedzie (np. nastąpi awaria węzła lub przekroczenie limitu czasu dla żądania), koordynator lub dowolny z uczestników może anulować transakcję.
3. Gdy aplikacja jest gotowa do zatwierdzenia transakcji, koordynator przesyła do wszystkich uczestników żądanie przygotowania się opatrzone globalnym identyfikatorem transakcji. Jeśli któreś z tych żądań nie powiedzie się lub spowoduje przekroczenie limitu czasu, koordynator wysyła do wszystkich uczestników żądanie anulowania dotyczące danego identyfikatora.
4. Gdy uczestnik otrzymuje żądanie przygotowania się, upewnia się, że niezależnie od okoliczności może zatwierdzić transakcję. W tym celu zapisuje wszystkie dane transakcji na dysku (awaria, brak zasilania lub wyczerpanie się miejsca na dysku nie są akceptowalnymi wymówkami do tego, by później odmówić zatwierdzenia) i sprawdza konflikty lub naruszenie więzów. Udzielenie odpowiedzi „tak” koordynatorowi oznacza, że węzeł obiecuje na żądanie zatwierdzić transakcję bez błędów. Tym samym uczestnik zrzeka się prawa do anulowania transakcji, ale jeszcze jej nie zatwierdza.
5. Gdy koordynator uzyskuje odpowiedzi na wszystkie żądania przygotowania się, podejmuje definitywną decyzję dotyczącą tego, czy zatwierdzić transakcję, czy ją anulować (zatwierdzenie następuje tylko, jeśli wszyscy uczestnicy odpowiedzieli „tak”). Koordynator musi zapisać tę decyzję w dzienniku transakcji na dysku, aby zapamiętać tę decyzję na wypadek późniejszej awarii. Jest to tzw. *punkt zatwierdzania*.
6. Po zapisaniu decyzji koordynatora na dysku żądanie zatwierdzenia lub anulowania jest przesyłane do wszystkich uczestników. Jeśli to żądanie kończy się niepowodzeniem lub przekroczeniem limitu czasu, koordynator musi w nieskończoność ponawiać próby. Nie ma drogi odwrotu. Jeśli zdecydowano się zatwierdzić transakcję, trzeba to wymusić niezależnie od liczby prób. Jeśli uczestnik w międzyczasie uległ awarii, transakcja zostanie zatwierdzona po przywróceniu stanu tego węzła. Ponieważ uczestnik zagłosował na „tak”, po wznowieniu pracy nie może odmówić zatwierdzenia transakcji.

Ten protokół obejmuje więc dwa kluczowe „punkty bez odwrotu”. Gdy uczestnik głosuje na „tak”, obiecuje, że z pewnością będzie mógł zatwierdzić transakcję (choć koordynator nadal może postanowić o jej anulowaniu). Ponadto gdy koordynator podejmuje decyzję, jest ona nieodwracalna. Te obietnice gwarantują atomowość dwuetapowego zatwierdzania. W atomowym zatwierdzaniu w jednym węźle te dwa zdarzenia są łączone w jedno: zapis rekordu zatwierdzającego w dzienniku transakcji.

Wróćmy do analogii z małżeństwem. Zanim powiesz „tak”, Ty i Twoja partnerka lub Twój partner możecie „anulować transakcję”, mówić „nigdy w życiu!” (lub coś podobnego). Jednak gdy powiesz „tak”, nie możesz już tego cofnąć. Jeśli później zemdlejesz i nie usłyszysz, jak duchowny mówi: „Ogłaszam was mężem i żoną”, nie zmienia to faktu, że transakcja została zatwierdzona. Gdy odzyskasz świadomość, możesz się dowiedzieć, czy zawarłeś związek małżeński, pytając duchownego o status swojego globalnego identyfikatora transakcji. Możesz też poczekać do chwili, gdy duchowny ponowi żądanie zatwierdzenia (ponieważ próby są ponawiane w czasie, kiedy jesteś nieprzytomny).

### **Awaria koordynatora**

Opisano już, co się dzieje, gdy jeden z uczestników lub sieć ulega awarii w trakcie zatwierdzania dwuetapowego. Jeśli jedno z żądań przygotowania się nie powiedzie lub przekroczy limit czasu, koordynator anuluje transakcję. Z kolei niepowodzenie żądań zatwierdzenia lub anulowania skutkuje ponawianiem prób w nieskończoność przez koordynatora. Mniej jasne jest jednak to, co się dzieje po awarii koordynatora.

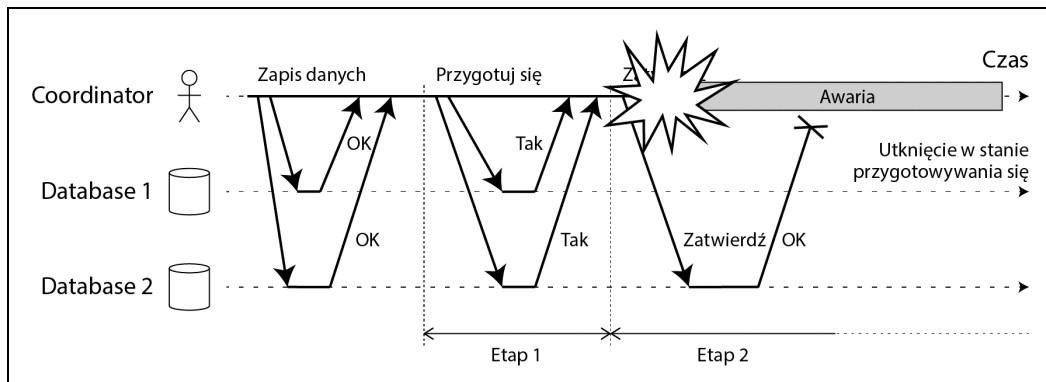
Jeśli koordynator ulegnie awarii przed przesłaniem żądania przygotowania się, uczestnik może bezpiecznie anulować transakcję. Jednak gdy uczestnik otrzymał żądanie przygotowania się i zagłosował na „tak”, nie może już jednostronnie anulować transakcji. Musi czekać na informację od koordynatora o tym, czy transakcja została zatwierdzona, czy odrzucona. Jeśli na tym etapie nastąpi awaria koordynatora lub sieci, uczestnik może tylko czekać. Transakcja uczestnika na tym etapie jest *nieokreślona*.

Tę sytuację ilustruje rysunek 9.10. W tym przykładzie koordynator decyduje się zatwierdzić transakcję, a baza 2 otrzymuje żądanie zatwierdzenia. Jednak przed przesłaniem żądania zatwierdzenia do bazy 1 następuje awaria koordynatora. Dlatego baza 1 nie wie, czy ma zatwierdzić transakcję, czy ją anulować. Nie pomaga tu nawet limit czasu. Jeśli baza 1 jednostronnie anuluje transakcję po upływie limitu czasu, znajdzie się w stanie niespójnym z bazą 2, która zatwierdziła operację. Podobnie niebezpieczne jest jednostronne zatwierdzanie transakcji, ponieważ inny uczestnik mógł ją anulować.

Bez informacji od koordynatora uczestnik nie wie, czy ma zatwierdzić transakcję, czy ją anulować. W zasadzie uczestnicy mogliby komunikować się między sobą, aby ustalić, jak głosował każdy z nich, i w ten sposób dojść do zgody. Jednak protokół dwuetapowego zatwierdzania tego nie obejmuje.

Jednym sposobem na zakończenie pracy protokołu dwuetapowego zatwierdzania jest odczekanie na wznowienie pracy przez koordynatora. To dlatego koordynator musi zapisywać decyzję o zatwierdzeniu lub anulowaniu w dzienniku transakcji na dysku przed przesłaniem żądań zatwierdzenia lub anulowania do uczestników. Gdy koordynator wznowi pracę, ustali status wszystkich





Rysunek 9.10. Awaria koordynatora po tym, jak uczestnicy zagłosowali na „tak”. Baza 1 nie wie, czy ma zatwierdzić transakcję, czy ją anulować

wątpliwych transakcji na podstawie dziennika. Wszystkie transakcje, dla których w dzienniku koordynatora nie ma rekordu zatwierdzającego, są anulowane. Dlatego punkt zatwierdzania w zatwierdzaniu dwuetapowym to odpowiednik zwykłego atomowego zatwierdzania w jednym węźle — w koordynatorze.

### Zatwierdzanie trzyetapowe

Zatwierdzanie dwuetapowe jest nazywane *blokującym* protokołem zatwierdzania atomowego, ponieważ może się zatrzymać w oczekiwaniu na wznowienie pracy przez koordynatora. Teoretycznie możliwe jest, by utworzyć *nieblokujący* protokół zatwierdzania atomowego, który nie zatrzymuje się po awarii węzła. Jednak w praktyce nie jest to takie proste.

Jako alternatywę dla zatwierdzania dwuetapowego zaproponowano algorytm *zatwierdzania trzyetapowego* [13, 80]. Przyjmuje się w nim jednak ograniczone opóźnienie i węzły z ograniczonymi czasami odpowiedzi. W większości stosowanych w praktyce systemów z nieograniczonym opóźnieniem sieciowym i przerwami w pracy procesów (zob. rozdział 8.) algorytm ten nie gwarantuje atomowości.

Nieblokujące zatwierdzanie atomowe wymaga *doskonałego wykrywacza awarii* [67, 71], czyli niezawodnego mechanizmu informowania, czy węzeł uległ awarii, czy nie. W sieci z nieograniczonym opóźnieniem limit czasu nie jest niezawodnym wykrywaczem awarii, ponieważ żądanie może przekroczyć ten limit z powodu problemów z siecią nawet wtedy, jeśli węzeł działa poprawnie. Dlatego nadal stosowane jest zatwierdzanie dwuetapowe mimo znanego problemu z awarią koordynatora.

## Transakcje rozproszone w praktyce

Transakcje rozproszone, zwłaszcza te implementowane za pomocą zatwierdzania dwuetapowego, są oceniane w różny sposób. Z jednej strony zapewniają ważne gwarancje bezpieczeństwa, które trudno jest inaczej uzyskać. Z drugiej strony są krytykowane za to, że powodują problemy operacyjne, obniżają wydajność i obiecują więcej, niż mogą zapewnić [81, 82, 83, 84]. Wiele usług działających w chmurze nie stosuje transakcji rozproszonych z powodu problemów operacyjnych związanych z takimi transakcjami [85, 86].

Niektóre implementacje transakcji rozproszonych powodują znaczny spadek wydajności. Na przykład transakcje rozproszone w bazie MySQL są ponad 10 razy wolniejsze niż transakcje w jednym węźle [87]. Nie jest więc zaskoczeniem, że ludzie odradzają ich stosowanie. Duża część spadku wydajności nieodłącznego od zatwierdzania dwuetapowego wynika z dodatkowego wymuszania zapisu na dysku za pomocą wywołania `fsync` (jest to potrzebne do odzyskiwania stanu po awarii [88]) i dodatkowego przesyłu komunikatów w sieci.

Jednak zamiast od razu rezygnować z transakcji rozproszonych, należy starannie je przeanalizować, ponieważ wiele można się z nich nauczyć. Przede wszystkim trzeba precyzyjnie określić, czym są „transakcje rozproszone”. Często mylone są ze sobą dwa stosunkowo odmienne rodzaje transakcji:

#### *Transakcje rozproszone wewnątrz bazy danych*

Niektóre rozproszone bazy danych (np. bazy używające replikacji i partycji w konfiguracji standardowej) obsługują transakcje wewnętrzne między węzłami. Na przykład bazy VoltDB i MySQL NDB Cluster udostępniają obsługę takich wewnętrznych transakcji. Wtedy wszystkie węzły uczestniczące w transakcji używają tego samego oprogramowania bazodanowego.

#### *Niejednorodnie transakcje rozproszone*

W transakcji *niejednorodnej* uczestnicy korzystają z przynajmniej dwóch różnych technologii. Mogą to być dwie bazy od różnych producentów, a nawet systemy inne niż bazy danych (np. brokery komunikatów). Transakcja rozproszona obejmująca takie systemy musi gwarantować zatwierdzanie atomowe nawet wtedy, gdy te systemy mogą działać zupełnie inaczej.

Transakcje wewnątrz bazy danych nie muszą być zgodne z żadnym innym systemem, dlatego można wtedy zastosować dowolny protokół i optymalizacje specyficzne dla konkretnej technologii. Z tego powodu transakcje rozproszone wewnątrz bazy danych często działają całkiem sprawnie. Z kolei transakcje z udziałem niejednorodnych technologii sprawiają znacznie więcej trudności.

#### **Przetwarzanie komunikatów tylko raz**

Niejednorodnie transakcje rozproszone umożliwiają integrowanie różnorodnych systemów w wartościowy sposób. Na przykład potwierdzenie przetworzenia komunikatu z kolejki może być przesyłane wtedy i tylko wtedy, jeśli transakcja bazodanowa przetwarzająca ten komunikat została z powodzeniem zatwierdzona. Można to zaimplementować, atomowo zatwierdzając w jednej transakcji potwierdzenie komunikatu i zapisy w bazie. Dzięki obsłudze transakcji rozproszonych jest to możliwe nawet wtedy, gdy broker komunikatów i baza są dwiema niepowiązanymi technologiami działającymi na różnych maszynach.

Jeśli nie powiedzie się albo dostarczenie komunikatu, albo transakcja w bazie, obie te operacje zostają anulowane, dlatego broker komunikatów może następnie bezpiecznie ponowić próbę przesyłania komunikatu. Tak więc dzięki atomowemu zatwierdzeniu komunikatu i efektów ubocznych jego przetwarzania można zagwarantować, że komunikat zostanie *skutecznie* przetworzony dokładnie raz — nawet jeśli wymaga to kilku prób. Anulowanie prowadzi do wycofania wszystkich efektów ubocznych częściowo ukończonej transakcji.

Tego rodzaju transakcje rozproszone są jednak możliwe tylko wtedy, gdy wszystkie systemy uwzględniane w transakcji mogą się posłużyć tym samym protokołem atomowego zatwierdzania.

Załóżmy np., że efekt uboczny przetwarzania komunikatu jest wysyłany na e-mail, a serwer poczty elektronicznej nie obsługuje zatwierdzania dwuetapowego. Jeśli przetwarzanie komunikatu się nie powiedzie i nastąpi ponowienie próby, może się zdarzyć, że e-mail zostanie przesłany przynajmniej dwa razy. Jeżeli jednak wszystkie efekty uboczne przetwarzania komunikatu zostaną wycofane po anulowaniu transakcji, etap przetwarzania można bezpiecznie ponowić, tak jakby nic się nie stało.

Do tematu przetwarzania komunikatów tylko raz wracamy w rozdziale 11. Najpierw jednak warto się przyjrzeć protokołowi zatwierdzania atomowego, który umożliwia działanie niejednorodnych transakcji rozproszonych.

## Transakcje XA

*X/Open XA* (ang. *eXtended Architecture*) to standard implementowania zatwierdzania dwuetapowego dla technologii niejednorodnych [76, 77]. Został wprowadzony w 1991 r. i jest powszechnie stosowany. Transakcje XA są obsługiwane w wielu tradycyjnych bazach relacyjnych (takich jak PostgreSQL, MySQL, DB2, SQL Server i Oracle) oraz brokerach komunikatów (takich jak ActiveMQ, HornetQ, MSMQ i IBM MQ).

XA nie jest protokołem sieciowym. Jest to interfejs API z języka C służący do komunikowania się z koordynatorem transakcji. W innych językach dostępne są mechanizmy komunikacji z tym interfejsem API. Na przykład w aplikacjach w Javie EE transakcje XA są implementowane za pomocą interfejsu **JTA** (ang. *Java Transaction API*), który z kolei jest obsługiwany przez wiele sterowników baz danych używających interfejsu **JDBC** (ang. *Java Database Connectivity*) i sterowników brokerów komunikatów używających interfejsu **JMS** (ang. *Java Message Service*).

W XA przyjmuje się, że aplikacja używa sterownika sieciowego lub biblioteki klienckiej do komunikowania się z bazami lub usługami obsługi komunikatów uczestniczącymi w transakcji. Jeśli sterownik obsługuje standard XA, oznacza to, że wywołuje interfejs API XA w celu ustalenia, czy operacja powinna być częścią transakcji rozproszonej. Jeżeli tak, sterownik przesyła niezbędne informacje na serwer bazodanowy. Sterownik udostępnia też wywołania zwrotne, za pomocą których koordynator może zażądać od uczestnika przygotowania się, zatwierdzenia transakcji lub jej anulowania.

Koordynator transakcji obejmuje implementację interfejsu API XA. Standard nie określa, jak ta implementacja powinna wyglądać. Jednak w praktyce koordynator jest często zwykłą biblioteką wczytywaną w tym samym procesie, w którym działa aplikacja uruchamiająca transakcję (koordynator nie jest odrębną usługą). Koordynator śledzi uczestników transakcji, rejestruje (za pomocą wywołań zwrotnych ze sterownika) ich odpowiedzi na żądanie przygotowania się i używa dziennika na dysku lokalnym do zapisywania decyzji o zatwierdzeniu lub anulowaniu poszczególnych transakcji.

Jeśli proces aplikacji ulegnie awarii lub maszyna, na której ta aplikacja jest uruchomiona, przestanie działać, to samo dotknie też koordynatora. Wszyscy uczestnicy, którzy się przygotowali, ale nie zatwierdzili transakcji, nie wiedzą, co mają zrobić. Ponieważ dziennik koordynatora znajduje się na dysku lokalnym serwera aplikacji, serwer trzeba ponownie uruchomić, a biblioteka koordynatora musi wczytać dziennik w celu odzyskania informacji o zatwierdzeniu lub anulowaniu

każdej transakcji. Dopiero po tym koordynator może za pomocą wywołań zwrotnych XA ze sterownika bazodanowego zażądać od uczestników zatwierdzenia lub anulowania transakcji. Serwer bazodanowy nie może bezpośrednio kontaktować się z koordynatorem, ponieważ cała komunikacja musi się odbywać poprzez bibliotekę kliencką.

### Utrzymywanie blokad, gdy występują wątpliwości

Dlaczego takim problemem jest utknięcie transakcji na etapie wątpliwości? Czy reszta systemu nie może kontynuować pracy, ignorując wątpliwą transakcję, która ostatecznie zostanie przetworzona?

Problem wynika z *blokad*. W punkcie „Odczyt zatwierdzonych danych” opisano, że transakcje w bazie zwykle zajmują blokadę na wyłączność na poziomie wiersza (dotyczy to wszystkich wierszy), aby zapobiec brudnym zapisom. Ponadto jeśli chcesz uzyskać izolację na poziomie sekwencyjności, baza używająca blokad dwuetapowych musi też zajmować blokadę współużytkowaną wszystkich wierszy *wczytywanych* przez transakcję (zob. punkt „Blokady dwuetapowe”).

Baza nie może zwolnić tych blokad do momentu zatwierdzenia lub anulowania transakcji (na rysunku 9.9 blokady są wyróżnione ciemniejszym kolorem). Dlatego w zatwierdzaniu dwuetapowym transakcja musi zajmować blokady do czasu wyeliminowania wątpliwości. Jeśli nastąpiła awaria koordynatora i jego ponowne uruchomienie zajmuje 20 minut, blokady są utrzymywane przez 20 minut. Jeżeli z jakiegoś powodu dziennik koordynatora został całkowicie utracony, blokady są zajmowane w nieskończoność — a przynajmniej do czasu ręcznego rozwiązania problemu przez administratora.

W trakcie utrzymywania blokad żadna inna transakcja nie może modyfikować wierszy. W zależności od bazy blokada może nawet uniemożliwiać innym transakcjom odczyt tych wierszy. Tak więc inne transakcje nie mogą po prostu kontynuować pracy. Jeśli zechcą uzyskać dostęp do tych samych danych, zostaną zablokowane. Może to spowodować, że duże części aplikacji staną się niedostępne do czasu zakończenia wątpliwej transakcji.

### Odyskiwanie stanu po awarii koordynatora

Teoretycznie jest tak, że po awarii i ponownym uruchomieniu koordynator powinien odtworzyć stan na podstawie dziennika i zakończyć wątpliwe transakcje. Jednak w praktyce zdarzają się *osierocone* wątpliwe transakcje [89, 90], czyli transakcje, dla których koordynator z jakiejś przyczyny nie może ustalić wyniku (np. z powodu utracenia dziennika transakcji lub jego uszkodzenia z powodu błędu oprogramowania). Takich transakcji nie da się automatycznie zamknąć, dlatego pozostają one w nieskończoność aktywne w bazie, zajmując blokady i blokując inne transakcje.

Nawet ponowne uruchomienie serwerów bazodanowych nie rozwiązuje tego problemu, ponieważ poprawna implementacja zatwierdzania dwuetapowego wymaga utrzymywania blokad przez wątpliwe transakcje nawet po restarcie (w przeciwnym razie występuje ryzyko naruszenia gwarancji atomowości). Jest to trwała sytuacja.

Jedynym rozwiązaniem jest ręczne zdecydowanie przez administratora o tym, czy zatwierdzić, czy anulować transakcje. Administrator musi sprawdzić uczestników każdej wątpliwej transakcji, ustalić, czy zatwierdzili oni lub anulowali transakcję, a następnie wprowadzić ten sam efekt u innych uczestników. Rozwiązanie problemu może wymagać znacznej ilości ręcznej pracy wykonywanej prawdopodobnie pod dużą presją stresu i czasu w trakcie poważnego przestoju środowiska produkcyjnego (w przeciwnym razie dlaczego koordynator byłby w tak złym stanie?).

W wielu implementacjach standardu XA stosowane jest wyjście awaryjne — *decyzje heurystyczne*. Umożliwia ono uczestnikom jednostronne decydowanie o anulowaniu lub zatwierdzeniu wątpliwej transakcji bez definitywnej decyzji koordynatora [76, 77, 91]. Warto zauważyć, że określenie *heurystyczne* jest tu eufemizmem oznaczającym *prawdopodobnie naruszające atomowość*, ponieważ naruszany jest system obietnic w zatwierdzaniu dwuetapowym. Dlatego decyzje heurystyczne są przeznaczone tylko do wydobywania się z katastrofalnych sytuacji, a nie do regularnego użytku.

## Ograniczenia transakcji rozproszonych

Transakcje XA rozwiązują rzeczywisty i ważny problem utrzymywania spójności kilku uczestników w systemie danych. Jednak, jak zobaczyłeś, powodują też poważne problemy operacyjne. Bardzo ważnym spostrzeżeniem jest to, że sam koordynator transakcji jest pewnego rodzaju bazą danych (w której zapisywane są wyniki transakcji). Dlatego trzeba go traktować równie ostrożnie jak inne ważne bazy:

- Jeśli dla koordynatora nie jest stosowana replikacja (działa on na jednej maszynie), stanowi on pojedynczy punkt podatności na awarię na poziomie całego systemu (ponieważ jego usterka powoduje, że inne serwery aplikacji są wstrzymywane przez blokady utrzymywane przez wątpliwe transakcje). Zaskakujące jest to, że wiele implementacji koordynatora nie zapewnia domyślnie wysokiej dostępności lub oferuje tylko podstawową obsługę replikacji.
- Wiele aplikacji działających po stronie serwera jest pisanych w modelu bezstanowym (faworyzowanym w HTTP), a cały stan trwały jest przechowywany w bazie. Zaletą tego modelu jest to, że serwery aplikacji można swobodnie dodawać i usuwać. Jednak gdy koordynator działa na serwerze aplikacji, zmienia to naturę systemu. Nagle dzienniki koordynatora stają się kluczową częścią trwałego stanu systemu — równie ważną jak same bazy, ponieważ dzienniki koordynatora są niezbędne do odzyskania wątpliwych transakcji po awarii. Takie serwery aplikacji nie są już bezstanowe.
- Ponieważ transakcje XA muszą być zgodne z wieloma systemami danych, z natury stanowią „najmniejszy wspólny mianownik”. Nie mogą np. wykrywać zakleszczeń na poziomie różnych systemów (ponieważ wymagałoby to standardowego protokołu wymiany informacji o blokadach, na które czekają poszczególne transakcje) i nie są zgodne z SSI (zob. punkt „Algorytm SSI”), ponieważ do tego niezbędny byłby protokół wykrywania konfliktów na poziomie różnych systemów.
- W wewnętrznych transakcjach rozproszonych (to nie transakcje XA) ograniczenia nie są tak poważne. Można np. zastosować rozproszoną wersję algorytmu SSI. Pozostaje jednak problem tego, że do udanego dwuetapowego zatwierdzenia niezbędne są odpowiedzi *wszystkich* uczestników. Dlatego jeśli *kórkolwiek* część systemu jest uszkodzona, transakcja zakończy się niepowodzeniem. Tak więc transakcje rozproszone *nasilają awarie*, co jest sprzeczne z celem budowania systemów odpornych na błędy.

Czy to oznacza, że należy porzucić nadzieję na utrzymanie spójności między kilkoma systemami? Nie do końca. Istnieją inne metody pozwalające uzyskać ten sam efekt bez problemów typowych dla niejednorodnych transakcji rozproszonych. Do tych tematów wrócimy w rozdziałach 11. i 12. Najpierw jednak należy podsumować zagadnienie konsensusu.

## Konsensus z zachowaniem odporności na błędy

Nieformalnie konsensus to efekt zgodzenia się kilku węzłów w jakiejś kwestii. Na przykład jeśli kilka osób próbuje jednocześnie zarezerwować ostatnie miejsce w samolocie lub teatrze albo zarejestrować konto o tej samej nazwie, algorytm osiągnięcia konsensusu powinien móc ustalić, która z tych wzajemnie wykluczających się operacji ma się powieść.

Problem osiągnięcia konsensusu jest zwykle formalnie ujmowany w następujący sposób: węzły mogą *zapropionować* wartości, a algorytm osiągnięcia konsensusu *decyduje* o wyborze jednej z nich. W przykładzie z rezerwowaniem miejsc w momencie, gdy kilku klientów próbuje jednocześnie kupić ostatni bilet, każdy węzeł obsługujący żądanie klienta może zaproponować identyfikator tego klienta. Decyzja określa wtedy, który z klientów otrzyma miejsce.

W tym formalnym ujęciu algorytm osiągnięcia konsensusu musi mieć następujące cechy [25]<sup>13</sup>:

### *Jednomyślna zgoda*

Żadne dwa węzły nie podejmują różnych decyzji.

### *Integralność*

Żaden węzeł nie podejmuje decyzji dwukrotnie.

### *Poprawność*

Jeśli węzeł wybiera wartość  $v$ ,  $v$  została zaproponowana przez któryś z węzłów.

### *Możliwość zakończenia*

Każdy węzeł, w którym nie nastąpiła awaria, ostatecznie decyduje się na którąś z wartości.

Jednomyślna zgoda i integralność definiują podstawową ideę konsensusu: wszyscy zgadzają się co do wyniku, a po jego ustaleniu nie można zmienić zdania. Poprawność ma przede wszystkim eliminować trywialne rozwiązania. Możesz np. utworzyć algorytm, który niezależnie od propozycji zawsze wybiera wartość  $nu11$ . Taki algorytm ma cechy jednomyślnej zgody i integralności, ale nie zapewnia poprawności.

Jeśli odporność na błędy nie ma dla Ciebie znaczenia, zapewnienie trzech pierwszych cech jest łatwe. Można zapisać na stałe, że jeden węzeł jest „dyktatorem”, i pozwolić mu na podejmowanie decyzji. Jeżeli jednak ten jeden węzeł przestanie działać, system nie będzie mógł podejmować żadnych decyzji. Tak właśnie działo się w zatwierdzaniu dwuetapowym — po awarii koordynatora uczestnicy mający wątpliwości nie mogli zdecydować, czy zatwierdzić operację, czy je anulować.

Cecha zakończenia to formalne ujęcie odporności na błędy. Zgodnie z tą cechą algorytm osiągnięcia konsensusu nie może w nieskończoność pozostawać nieaktywny. Oznacza to, że musi robić postępy. Nawet jeśli niektóre węzły dotknie awaria, pozostałe muszą podjąć decyzję. Możliwość zakończenia to cecha związana z aktywnością, natomiast pozostałe trzy właściwości są związane z bezpieczeństwem; zob. punkt „Bezpieczeństwo i aktywność”.

<sup>13</sup> Ta konkretna odmiana konsensusu jest nazywana *konsensusem jednogłośnym* (ang. *uniform consensus*) i odpowiada zwykłemu konsensusowi w systemach asynchronicznych z zawodnymi wykrywaczami awarii [71]. W literaturze naukowej częściej się pisze o *procesach* niż o *węzłach*, jednak tu, aby zachować spójność z resztą książki, stosowane jest pojęcie *węzły*.

W modelu systemu wykorzystującym konsensus przyjmuje się, że po awarii węzeł natychmiast znika i nigdy nie wraca. Wyobraź sobie, że zamiast awarii oprogramowania nastąpiło trzęsienie ziemi i centrum danych obejmujące dany węzeł zapadło się. Musisz przyjąć, że węzeł jest zakopany pod 10 metrami błota i nigdy nie wznowi pracy. W tym modelu systemu każdy algorytm, który musi czekać na przywrócenie stanu przez węzeł, nie będzie mógł spełnić wymogu możliwości zakończenia. Tego wymogu nie spełnia możliwość dwuetapowe zatwierdzanie.

Oczywiście jeśli awaria dotknie *wszystkie* węzły i ani jeden z nich nie będzie działał, żaden algorytm nie będzie mógł podjąć decyzji. Występuje limit awarii, jakie algorytm potrafi znieść. Można udowodnić, że każdy algorytm osiągnięcia konsensusu wymaga poprawnej pracy przynajmniej większości węzłów, aby zagwarantować możliwość zakończenia [67]. Ta większość może bezpiecznie stworzyć kworum (zob. punkt „Odczyty i zapisy zgodne z kworum”).

Tak więc cecha możliwości zakończenia wymaga przyjęcia założenia, że mniej niż połowa węzłów jest uszkodzona lub niedostępna. Jednak większość implementacji osiągnięcia konsensusu gwarantuje, że cechy związane z bezpieczeństwem (jednomyślność, integralność i poprawność) są spełnione zawsze, nawet gdy większość węzłów uległa awarii lub gdy występuje poważny problem z siecią [92]. Dlatego przestój na dużą skalę może uniemożliwić systemowi przetwarzanie żądań, ale nie powoduje, że system osiągnięcia konsensusu podejmie nieprawidłowe decyzje.

Większość algorytmów osiągnięcia konsensusu przyjmuje, że nie występują wady bizantyjskie (opisane w punkcie „Wady bizantyjskie”). Oznacza to, że jeśli węzeł nie przestrzega protokołu (np. przesyła sprzeczne komunikaty do różnych węzłów), może naruszyć cechy związane z bezpieczeństwem. Można zabezpieczyć mechanizm osiągnięcia konsensusu przed wadami bizantyjskimi, jeśli mniej niż jedna trzecia węzłów jest narażona na takie wady [25, 93]. Jednak w tej książce nie ma miejsca na szczegółowe omawianie takich algorytmów.

### **Algorytmy osiągnięcia konsensusu a rozgłaszanie z uporządkowaniem całkowitym**

Najbardziej znane odporne na błędy algorytmy osiągnięcia konsensusu to Viewstamped Replication (VSR) [94, 95], Paxos [96, 97, 98, 99], Raft [22, 100, 101] i Zab [15, 21, 102]. Występują między nimi pewne podobieństwa, jednak te algorytmy nie są identyczne [103]. W tej książce poszczególne algorytmy nie są szczegółowo omawiane. Wystarczy poznać wspólne im ogólne kwestie, chyba że sam chcesz zaimplementować system osiągnięcia konsensusu (co nie jest polecane; jest to trudne zadanie [98, 104]).

Większość wymienionych algorytmów nie wykorzystuje bezpośrednio opisanego tu formalnego modelu (z proponowaniem i wyborem jednej wartości przy zachowaniu cech jednomyślności, integralności, poprawności i możliwości zakończenia). Zamiast tego wybierana jest *sekwencja* wartości, są to więc algorytmy *rozgłaszania z uporządkowaniem całkowitym*, co opisano wcześniej w rozdziale (zob. punkt „Rozgłaszanie z uporządkowaniem całkowitym”).

Pamiętaj, że takie rozgłaszanie wymaga dostarczania komunikatów do wszystkich węzłów dokładnie raz i w tej samej kolejności. Jeśli się nad tym zastanowić, jest to odpowiednik kilku powtórzeń procesu osiągnięcia konsensusu. W każdym powtórzeniu węzły proponują komunikat, który chcą przesłać jako następny, po czym decydują o kolejnym komunikacie przesyłanym zgodnie z uporządkowaniem całkowitym [67].

Tak więc rozgłaszanie z uporządkowaniem całkowitym to odpowiednik powtórzeń procesu osiągnięcia konsensusu (każda decyzja zgodna z konsensusem odpowiada dostarczeniu jednego komunikatu):

- Dzięki zapewnianej w konsensusie jednomyślności wszystkie węzły decydują się dostarczać te same komunikaty w tej samej kolejności.
- Dzięki integralności komunikaty nie są duplikowane.
- Dzięki poprawności komunikaty nie są uszkodzone lub sfabrykowane.
- Dzięki możliwości zakończenia komunikaty nie giną.

W algorytmach VSR, Raft i Zab bezpośrednio zaimplementowane jest rozgłaszanie z uporządkowaniem całkowitym, ponieważ jest to wydajniejsze niż wiele powtórzeń osiągnięcia konsensusu dla pojedynczych wartości. W Paxosie taka optymalizacja nosi nazwę Multi-Paxos.

### **Replikacja z jednym liderem a konsensus**

W rozdziale 5. opisano replikację z jednym liderem (zob. punkt „Liderzy i obserwatorzy”), gdzie wszystkie zapisy trafiają do lidera i są wprowadzane w obserwatorach w tej samej kolejności, dzięki czemu repliki pozostają aktualne. Czy nie wygląda to jak rozgłaszanie z uporządkowaniem całkowitym? Dlaczego w rozdziale 5. nie trzeba było martwić się osiągnięciem konsensusu?

Odpowiedź wiąże się ze sposobem wyboru lidera. Jeśli lider jest ręcznie wybierany i konfigurowany przez pracowników działu operacyjnego, powstaje algorytm osiągnięcia konsensusu w wersji z „dyktatorem”. Tylko jeden węzeł może akceptować zapisy (czyli podejmować decyzje o kolejności zapisów w dzienniku replikacji), a jeśli przestanie on działać, system staje się niedostępny do zapisu do momentu, gdy operatorzy ręcznie skonfigurują inny węzeł jako lidera. Taki system może w praktyce działać dobrze, jednak nie zapewnia możliwości zakończenia, ponieważ wymaga interwencji człowieka w celu zrobienia postępów.

Niektóre bazy automatycznie obsługują wybór lidera i przełączanie awaryjne, promując obserwatora na nowego lidera po awarii dawnego (zob. punkt „Radzenie sobie z awariami węzła”). Przybliża to do uzyskania odpornego na błędy rozgłaszania z uporządkowaniem całkowitym, a tym samym do rozwiązania problemu konsensusu.

Występuje jednak pewna trudność. Wcześniej opisano problem „rozszczenia mózgu” i stwierdzono, że wszystkie węzły muszą się zgadzać co do tego, kto jest liderem. W przeciwnym razie dwa różne węzły mogą się uważać za lidera i spowodować, że baza będzie w niespójnym stanie. Dlatego potrzebny jest konsensus w celu wyboru lidera. Jeśli jednak opisane tu algorytmy osiągnięcia konsensusu to algorytmy rozgłaszania z uporządkowaniem całkowitym, które z kolei działają jak replikacja z jednym liderem, a replikacja z jednym liderem wymaga lidera, to...

Wygląda na to, że aby móc wybrać lidera, najpierw potrzebny jest lider. W celu rozwiązania problemu osiągnięcia konsensusu, najpierw trzeba rozwiązać ten problem. Jak wyjść z tego błędnego kręgu?

### **Numerowanie epok i kwora**

Wszystkie opisane do tej pory protokoły osiągnięcia konsensusu wewnętrznie korzystają z lidera (w różnych postaciach), ale nie gwarantują, że lider jest unikatowy. Zamiast tego dają słabszą



gwarancję. Protokoły definiują *numer epoki* (w Paxosie nazywany *ballot number*, w VSR — *view number*, a w Rafcie — *term number*) i gwarantują, że lider jest unikatowy w ramach każdej epoki.

Za każdym razem, gdy lider jest uznawany za niesprawnego, rozpoczyna się głosowanie między węzłami w celu wyboru nowego lidera. Wynikowi przypisuje się zwiększony numer epoki, dlatego te numery mają uporządkowanie całkowite i są monotonicznie rosnące. Jeśli występuje konflikt między dwoma liderami z różnych epok (np. dlatego że wcześniejszy lider wcale nie był „martwy”), lider z wyższym numerem epoki jest uznawany za ważniejszego.

Zanim lider będzie mógł podjąć jakąkolwiek decyzję, musi najpierw sprawdzić, czy nie istnieje inny lider o wyższym numerze epoki, który mógłby zdecydować inaczej. Skąd lider ma wiedzieć, że nie został odsunięty przez inny węzeł? Przypomnij sobie punkt „Prawda jest definiowana przez większość”. Węzeł nie zawsze może ufać własnym sądom. Samo to, że węzeł uważa siebie za lidera, nie oznacza jeszcze, że inne węzły akceptują jego rolę.

Zamiast tego lider musi uzyskać głosy od *kworum* węzłów (zob. punkt „Odczyty i zapisy zgodne z kworum”). Lider w przypadku każdej decyzji, jaką chce podjąć, musi przesłać proponowaną wartość do innych węzłów i poczekać na to, aż kworum węzłów zaakceptuje propozycję. Zwykle (choć nie zawsze) kworum składa się z większości węzłów [105]. Węzeł głosuje na rzecz propozycji tylko wtedy, jeśli nie zna innego lidera o wyższym numerze epoki.

Występują więc dwie serie głosowań. W jednej wybierany jest lider, a drugiej węzły głosują nad propozycją lidera. Ważne jest to, że kwora w obu głosowaniach muszą choć częściowo się pokrywać. Jeśli głosowanie nad propozycją kończy się powodzeniem, przynajmniej jeden z węzłów głosujących nad nią musiał też uczestniczyć w ostatnich wyborach lidera [105]. Dlatego jeżeli w głosowaniu nad propozycją nie ujawniono epoki o wyższym numerze, aktualny lider może uznać, że nie nastąpił wybór lidera o wyższym numerze epoki. Stąd lider wie, że wciąż zajmuje to stanowisko. Pozwala mu to bezpiecznie podjąć decyzję w kwestii proponowanej wartości.

Ten proces głosowania pozornie jest podobny do zatwierdzania dwuetapowego. Największa różnica polega na tym, że w zatwierdzaniu dwuetapowym koordynator nie jest wybierany, a odporne na błędy algorytmy osiągania konsensusu wymagają tylko głosów od większości węzłów, podczas gdy w zatwierdzaniu dwuetapowym niezbędny jest głos na „tak” od *każdego* uczestnika. Ponadto algorytmy osiągania konsensusu definiują proces odzyskiwania stanu, dzięki któremu węzły mogą uzyskać spójny stan po wyborze nowego lidera. Gwarantuje to, że cechy związane z bezpieczeństwem zawsze obowiązuja. Te różnice są kluczowe w kontekście poprawności o odporności na błędy algorytmu osiągania konsensusu.

## Ograniczenia algorytmów osiągania konsensusu

Algorytmy osiągania konsensusu są istotnym przełomem w świecie systemów rozproszonych. Zapewniają konkretne cechy związane z bezpieczeństwem (jednomysłność, integralność i poprawność) w systemach, w których wszystko inne jest niepewne. Jednocześnie zachowują odporność na błędy (możliwość czynienia postępów, dopóki większość węzłów działa i jest dostępna). Zapewniają rozgłaszanie z uporządkowaniem całkowitym, dlatego pozwalają implementować liniowe operacje atomowe w sposób odporny na błędy (zob. punkt „Implementowanie baz liniowych za pomocą rozgłaszania z uporządkowaniem całkowitym”). Mimo to te algorytmy nie zawsze są stosowane, ponieważ zapewniane korzyści wiążą się z kosztami.

Proces głosowania przez węzły nad propozycjami przed podjęciem decyzji jest pewnego rodzaju replikacją synchroniczną. W punkcie „Replikacja synchroniczna i asynchroniczna” opisano, że bazy są często tak konfigurowane, by posługiwały się replikacją asynchroniczną. W tej konfiguracji część zatwierdzonych danych może zostać utracona w trakcie przełączania awaryjnego. Jednak wiele osób akceptuje to ryzyko, aby uzyskać wyższą wydajność.

Systemy oparte na konsensusie zawsze wymagają do działania większości węzłów. To oznacza, że potrzebne są przynajmniej trzy węzły, aby móc tolerować jedną awarię (pozostałe dwa z trzech węzłów tworzą większość), a uzyskanie odporności na dwie awarie wymaga minimum pięciu węzłów (pozostałe trzy z pięciu węzłów tworzą większość). Jeśli awaria sieci odcina część węzłów od pozostałych, tylko część sieci obejmująca większość jednostek może robić postępy. Reszta węzłów jest zablokowana (zob. też punkt „Koszty liniowości”).

Większość algorytmów osiągania konsensusu przyjmuje, że w głosowaniu uczestniczy stały zbiór węzłów. Oznacza to, że nie można swobodnie dodawać lub usuwać węzłów w klastrze. Rozszerzenia algorytmów obsługujące *dynamiczne członkostwo* umożliwiają modyfikowanie zbioru węzłów w klastrze w czasie, są jednak znacznie mniej poznane niż algorytmy ze statycznym członkostwem.

Algorytmy osiągania konsensusu zwykle wykorzystują limitu czasu do wykrywania niesprawnych węzłów. W środowiskach z wysoce zmiennymi opóźnieniami sieciowymi, a zwłaszcza w geograficznie rozproszonych systemach, często się zdarza, że węzeł z powodu przejściowych problemów z siecią fałszywie przyjmuje, iż lider uległ awarii. Choć ten błąd nie narusza cech związanych z bezpieczeństwem, częste wybory lidera skutkują bardzo niską wydajnością, ponieważ system może przeznaczać więcej czasu na wybieranie lidera niż na wykonywanie użytecznych zadań.

Czasem algorytmy osiągania konsensusu są wysoce wrażliwe na problemy z siecią. Wykazano np., że w algorytmie Raft występują nieprzyjemne przypadki brzegowe [106]. Jeśli cała sieć działa poprawnie z wyjątkiem jednego połączenia, które regularnie zawodzi, stanowisko lidera może być nieustannie przekazywane między dwoma węzłami lub aktualny lider jest stale zmuszany do rezygnacji. Dlatego system w zasadzie nie może robić postępów. Inne algorytmy osiągania konsensusu są narażone na podobne trudności, a zaprojektowanie algorytmów bardziej odpornych na zawodne sieci wciąż jest otwartym problemem naukowym.

## Usługi zarządzania członkostwem i koordynacją

Projekty takie jak ZooKeeper i etcd są często opisywane jako „rozproszone bazy z kluczami i wartościami” lub „usługi odpowiedzialne za koordynację i konfigurację”. Interfejs API takiej usługi wygląda podobnie jak interfejs bazy danych. Można wczytywać i zapisywać wartości dla danych kluczy oraz iteracyjnie sprawdzać klucze. Skoro więc są to zasadniczo bazy danych, po co kłopotać się implementowaniem algorytmu osiągania konsensusu? Co sprawia, że różnią się one od innych rodzajów baz?

Aby to zrozumieć, warto pokrótce przyjrzeć się temu, jak usługi takie jak ZooKeeper są stosowane. Programista aplikacji rzadko musi bezpośrednio korzystać z ZooKeepera, ponieważ ta usługa nie sprawdza się dobrze jako baza danych do użytku ogólnego. Bardziej prawdopodobne jest, że będzie korzystał z niej pośrednio za pomocą innego projektu. Na przykład HBase, Hadoop YARN, OpenStack Nova i Kafka wymagają działającego w tle ZooKeepera. Jakie korzyści mają z niego te projekty?

ZooKeeper i etcd są zaprojektowane do przechowywania niewielkich ilości danych, które w całości mieszczą się w pamięci (choć i tak są zapisywane na dysku, aby zapewnić im trwałość). Dlatego nie warto przechowywać w tych usługach wszystkich danych aplikacji. Ta niewielka ilość danych jest replikowana we wszystkich węzłach z użyciem odpornego na błędy algorytmu rozgłaszania z uporządkowaniem całkowitym. Wcześniej opisano, że takie rozgłaszanie to technika potrzebna do replikacji baz. Jeśli każdy komunikat reprezentuje zapis w bazie, wprowadzenie tych samych zapisów w tej samej kolejności sprawia, że repliki są spójne ze sobą.

ZooKeeper jest wzorowany na usłudze obsługi blokad Chubby firmy Google [14, 98], która udostępnia nie tylko rozgłaszanie z uporządkowaniem całkowitym (a tym samym i osiąganie konsensusu), ale też ciekawy zestaw innych cech, które okazują się wyjątkowo przydatne w trakcie budowania systemów rozproszonych:

#### *Liniowe operacje atomowe*

Za pomocą atomowej operacji „porównaj i ustaw” możesz zaimplementować blokadę. Jeśli kilka węzłów próbuje jednocześnie wykonać tę samą operację, tylko jednemu z nich to się uda. Protokół osiągania konsensusu gwarantuje, że operacja będzie atomowa i liniowa, nawet jeśli w dowolnym momencie węzeł zawiedzie lub nastąpi zakłócenie w pracy sieci. Blokadę rozproszone zwykle implementuje się w formie *dzierżaw*, które mają czas wygasania, dlatego są ostatecznie zwalniane, jeśli nastąpi awaria klienta (zob. punkt „Wstrzymywanie procesów”).

#### *Uporządkowanie całkowite operacji*

W punkcie „Lider a blokad” opisano, że gdy jakiś zasób jest chroniony za pomocą blokad lub dzierżawy, potrzebny jest *token odgradzający*, aby zapobiegać konfliktom między klientami w czasie wstrzymania pracy procesu. Token odgradzający to liczba rosnąca monotonicznie za każdym razem, gdy blokada jest zajmowana. ZooKeeper obsługuje tę technikę, zapewniając uporządkowanie całkowite wszystkich operacji i przypisując każdej z nich monotonicznie rosnący identyfikator (zxid) i numer wersji (cversion) [15].

#### *Wykrywanie awarii*

Klienty przechowują długie sesje na serwerach ZooKeepera. Klient i serwer okresowo przesyłają sygnały kontrolne, sprawdzając, czy drugi węzeł wciąż działa. Nawet jeśli połączenie zostanie tymczasowo zerwane lub węzeł z ZooKeeperem przestanie działać, sesja pozostanie aktywna. Jeśli jednak przerwa w przesyłaniu sygnałów kontrolnych jest dłuższa niż czas wygasania sesji, ZooKeeper uznaje sesję za zerwaną. Wszystkie blokad zajmowane przez sesję można tak skonfigurować, aby były automatycznie zwalniane po przekroczeniu limitu czasu (w ZooKeeperze takie węzły nazywa się *efemerycznymi*).

#### *Powiadomienia o zmianach*

Klient może nie tylko sprawdzać blokad i wartości innego klienta, ale też obserwować je pod kątem zmian. Dlatego klient może ustalić moment, w którym inny klient dołącza do klastra (na podstawie wartości zapisanej w ZooKeeperze) lub przestaje działać (ponieważ upłynął limit czasu dla sesji i efemeryczne węzły zniknęły). Dzięki subskrypcji powiadomień klienty unikają konieczności częstego zgłaszania zapytań w celu dowiedzenia się o zmianach.

Z tych mechanizmów tylko liniowe operacje atomowe rzeczywiście wymagają konsensusu. Jednak to połączenie opisanych cech sprawia, że systemy takie jak ZooKeeper są tak przydatne do koordynacji pracy w środowisku rozproszonym.

### Przydział zadań do węzłów

Jedną z sytuacji, gdzie model z narzędzi ZooKeeper i Chubby dobrze się sprawdza, jest używanie kilku instancji procesu lub usługi, z których jedną trzeba wybrać liderem. Jeśli lider zawiedzie, jeden z innych węzłów powinien przejąć jego zadania. Jest to oczywiście przydatne w bazach z jednym liderem, ale też w programach szeregujących i podobnych systemach stanowych.

Inny przykład to podzielony zasób (baza, strumień komunikatów, magazyn plików, system rozproszony z aktorami itd.) wymagający określenia, które partycje przypisać do poszczególnych węzłów. Gdy nowe węzły dołączają do klastra, niektóre partycje trzeba przenieść z istniejących węzłów do nowych, aby zrównoważyć obciążenie (zob. punkt „Równoważenie partycji”). Gdy węzły są usuwane lub przestają działać, inne jednostki muszą przejąć zadania uszkodzonych węzłów.

Tego rodzaju mechanizmy można uzyskać dzięki sensownemu wykorzystaniu operacji atomowych, węzłów efemerycznych i powiadomień w ZooKeeperze. Jeśli poprawnie zastosujesz to podejście, aplikacja będzie mogła automatycznie (bez interwencji człowieka) wznawiać pracę po awariach. Nie jest to łatwe — i to mimo dostępności bibliotek takich jak Apache Curator [17] mających zapewniać wysokopoziomowe narzędzia oparte na klienckim interfejsie API ZooKeepera. Jest to jednak znacznie lepsze rozwiązanie niż próba implementowania potrzebnych algorytmów osiągnięcia konsensusu od podstaw, która rzadko kończy się sukcesem [107].

Aplikacja początkowo może działać tylko w jednym węźle, ale ostatecznie rozrosnąć się do tysięcy węzłów. Próba głosowań z udziałem większości z tak wielu węzłów byłaby wysoce niewydajna. Zamiast tego ZooKeeper działa dla stałej liczby węzłów (zwykle trzech lub pięciu) i przeprowadza głosowania między tymi węzłami, a jednocześnie obsługuje potencjalnie dużą liczbę klientów. ZooKeeper umożliwia więc „outsourcing” części pracy związanej z koordynacją węzłów (osiągania konsensusu, porządkowania operacji i wykrywania awarii) do zewnętrznej usługi.

Dane zarządzane przez ZooKeepera zwykle rzadko się zmieniają. Reprezentują one informacje takie jak: „węzeł działający pod adresem 10.1.1.23 jest liderem partycji 7”, które mogą się zmieniać w skali minut lub godzin. ZooKeeper nie jest przeznaczony do przechowywania stanu wykonywania aplikacji, który może się zmieniać tysiące, a nawet miliony razy na sekundę. Jeśli stan aplikacji trzeba replikować między węzłami, można zastosować inne narzędzia (takie jak Apache BookKeeper [108]).

### Wykrywanie usług

ZooKeeper, etcd i Consul są często używane do *wykrywania usług*, czyli do ustalania, z którym adresem IP należy się połączyć, aby dotrzeć do konkretnej usługi. W centrach danych udostępniających chmurę, gdzie maszyny wirtualne często stale są uruchamiane i wyłączane, początkowo adresy IP usług często nie są znane. Zamiast tego można skonfigurować usługi w taki sposób, aby w momencie rozruchu rejestrowały sieciowe punkty końcowe w rejestrze usług, gdzie mogą być znalezione przez inne jednostki.

Jednak mniej jasne jest to, czy wykrywanie usług wymaga konsensusu. Tradycyjnym narzędziem do wyszukiwania adresu IP na podstawie nazwy usługi jest DNS. Aby zapewnić wysoką wydajność i dostępność, DNS wykorzystuje wiele warstw pamięci podręcznej. Odczyty z systemu DNS w żadnym razie nie są liniowe i to, że wyniki dla zapytań do tego systemu są trochę nieaktualne, zwykle nie jest uznawane za problem [109]. Ważniejsze jest to, że DNS zapewnia stabilną dostępność i jest odporny na zakłócenia w pracy sieci.

Choć wykrywanie usług nie wymaga konsensusu, jest on niezbędny przy wyborze lidera. Dlatego jeśli system osiągnięcia konsensusu wie już, kto jest liderem, sensowne może być wykorzystanie tej informacji do pomocy innym usługom w ustaleniu lidera. W tym celu niektóre systemy osiągnięcia konsensusu obsługują przeznaczone tylko do odczytu repliki z pamięcią podręczną. Te repliki asynchronicznie pobierają dziennik z wszystkimi decyzjami algorytmu osiągnięcia konsensusu, ale nie uczestniczą aktywnie w głosowaniu. Dlatego mogą obsługiwać żądania odczytu, które nie muszą być liniowe.

### Usługi związane z członkostwem

ZooKeeper i pokrewne narzędzia można traktować jako część długiej historii badań nad *usługami związanymi z członkostwem*. Te badania są prowadzone od lat 80. i były ważne w kontekście budowania wysoce niezawodnych systemów służących np. do kontroli ruchu lotniczego [110].

Usługa związana z członkostwem określa, które węzły są aktualnie aktywnymi i działającymi członkami klastra. Z rozdziału 8. dowiedziałeś się, że z powodu nieograniczonych opóźnień w sieci nie da się niezawodnie wykryć, czy inny węzeł przestał działać. Jeśli jednak połączyć wykrywanie awarii z konsensem, węzły mogą uzgodnić, które jednostki należy uznawać za aktywne, a które nie.

Może się zdarzyć, że węzeł zostanie zgodnie z konsensem mylnie uznany za „martwy”, choć w rzeczywistości działa. Jednak w systemie bardzo przydatna jest możliwość uzgodnienia, które węzły są jego aktualnymi członkami. Na przykład wybór lidera może polegać na wskazaniu węzła o najniższym numerze spośród aktualnych członków. To podejście nie zadziała jednak, jeśli różne węzły będą miały rozbieżne opinie na temat tego, kim są aktualni członkowie.

## Podsumowanie

W tym rozdziale z kilku różnych perspektyw omówiono tematy spójności i konsensusu. Opisano szczegółowo popularny model spójności, jakim jest liniowość. Ma ona sprawiać, by replikowane dane wyglądały tak, jakby występowały w jednym egzemplarzu, oraz by wszystkie operacje na nich przebiegały atomowo. Choć liniowość jest atrakcyjna, ponieważ łatwo ją zrozumieć (baza działa wtedy jak zmienna w programie jednowątkowym), wadą tego modelu jest powolność — zwłaszcza w środowiskach z wysokim opóźnieniem sieciowym.

Ponadto omówiono przyczynowość, która wymusza uporządkowanie zdarzeń w systemie (na podstawie przyczyn i skutków określa, co przed czym się zdarzyło). W odróżnieniu od liniowości porządkującej wszystkie operacje na jednej osi czasu z uporządkowaniem całkowitym przyczynowość oferuje słabszy model spójności. Niektóre zdarzenia mogą być jednoczesne, dlatego historia wersji przypomina oś czasu, która rozgałęzia się i schodzi. Spójność przyczynowa nie powoduje kosztów koordynacji typowych dla liniowości i jest znacznie mniej podatna na problemy z siecią.

Jednak nawet po wprowadzeniu uporządkowania przyczynowego (np. za pomocą znaczników czasu Lamporta) okazuje się, że niektórych rozwiązań nie da się zaimplementować. W punkcie „Uporządkowanie na podstawie znaczników czasu nie jest wystarczające” przeanalizowano za-pewnianie unikatowości nazwy użytkownika i odrzucanie jednoczesnych rejestracji tej samej nazwy. Jeśli jeden węzeł zamierza zaakceptować rejestrację, musi skądś wiedzieć, że inny węzeł jednocześnie nie rejestruje tej samej nazwy. Ten problem prowadzi do *konsensusu*.

Zobaczyłeś, że osiągnięcie konsensusu oznacza decydowanie o czymś w taki sposób, że wszystkie węzły zgadzają się co do decyzji i że jest ona nieodwołalna. Po analizach okazuje się, że wiele problemów można sprowadzić do osiągnięcia konsensusu i że są one ekwiwalentne względem siebie (w tym sensie, że rozwiązanie dla jednego z nich można łatwo przekształcić w rozwiązanie dla innych). Oto te problemy:

#### *Linijowe rejestry z operacją „porównaj i ustaw”*

Rejestr musi atomowo *decydować*, czy ustawić wartość. Decyzja jest podejmowana na podstawie tego, czy aktualna wartość rejestru jest równa parametrowi podanemu w operacji.

#### *Atomowe zatwierdzanie transakcji*

Baza musi decydować, czy zatwierdzić, czy anulować rozproszoną transakcję.

#### *Rozgłaszanie z uporządkowaniem całkowitym*

System obsługi komunikatów musi *decydować* o kolejności dostarczania komunikatów.

#### *Blokady i dzierżawy*

Gdy kilka klientów ściga się, by zająć blokadę lub dzierżawę, blokada *decyduje*, który z klientów ją uzyska.

#### *Usługi zarządzania członkostwem i koordynacją*

Na podstawie wykrywacza awarii (np. limitów czasu) system musi *decydować*, które węzły są aktywne, a które należy uznać za „martwe”, ponieważ w ich sesjach przekroczony został limit czasu.

#### *Więzy unikatowości*

Gdy kilka transakcji próbuje jednocześnie utworzyć sprzeczne rekordy dla tego samego klucza, więzy muszą *decydować*, której transakcji pozwolić wykonać zadanie, a które powinny zakończyć się niepowodzeniem z powodu naruszenia więzów.

Wszystkie te zadania są proste, jeśli używany jest tylko jeden węzeł lub gdy chcesz przydzielić zadanie podejmowania decyzji jednemu węzłowi. Tak jest w bazie z jednym liderem. Prawo do podejmowania decyzji jest przekazywane liderowi. To dlatego takie bazy mogą udostępniać operacje liniowe, więzy unikatowości, dziennik replikacji z uporządkowaniem całkowitym itd.

Jeśli jednak pojedynczy lider przestanie działać lub problemy z siecią sprawią, że stanie się on niedostępny, system nie będzie mógł zrobić żadnych postępów. Z taką sytuacją można sobie radzić na trzy sposoby:

1. Poczekać na wznowienie pracy przez lidera i zaakceptować to, że w międzyczasie system będzie zablokowany. Wiele koordynatorów transakcji XA i JTA działa w ten sposób. To podejście nie rozwiązuje w pełni problemu osiągnięcia konsensusu, ponieważ nie zapewnia możliwości zakończenia transakcji. Jeśli lider nie wznowi pracy, system może zostać trwale zablokowany.
2. Spróbować ręcznego przełączania awaryjnego, kiedy to człowiek wybiera nowego lidera i rekonfiguruje system. To podejście jest stosowane w wielu bazach relacyjnych. Jest to rodzaj konsensusu wprowadzanego przez „siłę wyższą”. To ludzki operator (spoza systemu informacyjnego) podejmuje decyzję. Szybkość przełączania awaryjnego jest ograniczona przez szybkość działania człowieka, który jest zwykle wolniejszy niż komputery.
3. Posłużyć się algorytmem do automatycznego wyboru nowego lidera. To podejście wymaga algorytmu osiągnięcia konsensusu. Zaleca się wtedy stosowanie sprawdzonego algorytmu, który poprawnie radzi sobie z niekorzystnymi warunkami w sieci [107].

Choć baza z jednym liderem pozwala zapewnić liniowość bez korzystania przy każdym zapisie z algorytmu osiągnięcia konsensusu, i tak wymaga konsensusu do określania lidera oraz przy jego zmianie. Dlatego używanie lidera jest w pewnym sensie „zamiataniem śmieci pod dywan”. Konsensus i tak jest konieczny, tylko w innym miejscu i rzadziej. Dobra wiadomość jest taka, że istnieją odporne na błędy algorytmy i systemy osiągnięcia konsensusu, pobeżnie omówione w tym rozdziale.

Narzędzia takie jak ZooKeeper odgrywają istotną rolę w „outsourcingu” osiągnięcia konsensusu, wykrywania błędów i zarządzania członkostwem w sposób, z którego mogą korzystać aplikacje. Używanie takich narzędzi nie jest łatwe, jednak i tak znacznie prostsze niż próba opracowania własnych algorytmów odpornych na wszystkie problemy opisane w rozdziale 8. Jeśli będziesz chciał wykonać jedno z zadań redukowalnych do problemu osiągnięcia konsensusu i zapewnić przy tym odporność na błędy, powinieneś zastosować rozwiązanie w rodzaju ZooKeepera.

Jednak nie każdy system wymaga konsensusu. Na przykład systemy z replikacją bez lidera lub z wieloma liderami zwykle nie wymagają globalnego konsensusu. Konflikty, które występują w takich systemach (zob. punkt „Radzenie sobie z konfliktami przy zapisie”), są konsekwencją braku konsensusu między różnymi liderami, jednak możliwe, że nie stanowi to problemu. Możliwe, że po prostu trzeba sobie radzić bez liniowości i nauczyć się lepiej posługiwać danymi z rozgałęziającą się i łączącą historią wersji.

W tym rozdziale wspomniano o wielu badaniach z dziedziny teorii systemów rozproszonych. Choć artykuły i dowody teoretyczne nie zawsze łatwo jest zrozumieć, a ponadto czasem przyjmowane są w nich nierealistyczne założenia, badania są niezwykle przydatnym źródłem wiedzy dla praktycznych prac w tej dziedzinie. Pomagają zrozumieć, co jest możliwe, a czego nie da się zrobić, oraz ułatwiają znajdowanie nieintuicyjnych błędów w systemach rozproszonych. Jeśli masz czas, warto się zapoznać z literaturą cytowaną.

To prowadzi do końca części II tej książki, gdzie omówiono replikację (rozdział 5.), podział na partycje (rozdział 6.), transakcje (rozdział 7.), modele błędów w systemach rozproszonych (rozdział 8.), a na zakończenie spójność i konsensus (rozdział 9.). Po przedstawieniu solidnych podstaw teoretycznych w części III ponownie wracamy do bardziej praktycznych systemów i wyjaśniamy, jak budować wartościowe aplikacje z niejednorodnych komponentów.

## Literatura cytowana

- [1] Peter Bailis i Ali Ghodsi, *Eventual Consistency Today: Limitations, Extensions, and Beyond*, „ACM Queue”, rocznik 11, nr 3, s. 55 – 63, marzec 2013 (<http://queue.acm.org/detail.cfm?id=2462076>; <https://dl.acm.org/citation.cfm?doid=2460276.2462076>).
- [2] Prince Mahajan, Lorenzo Alvisi i Mike Dahlin, *Consistency, Availability, and Convergence*, University of Texas at Austin, Department of Computer Science, raport techniczny UTCS TR-11-22, maj 2011 ([https://apps.cs.utexas.edu/tech\\_reports/reports/tr/TR-2036.pdf](https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2036.pdf)).
- [3] Alex Scotti, *Adventures in Building Your Own Database*, w: „All Your Base”, listopad 2015.
- [4] Peter Bailis, Aaron Davidson, Alan Fekete i in., *Highly Available Transactions: Virtues and Limitations*, w: „40th International Conference on Very Large Data Bases” (VLDB), wrzesień 2014 (<https://arxiv.org/pdf/1302.0309.pdf>). Rozszerzona wersja opublikowana jako preprint arXiv:1302.0309 [cs.DB].
- [5] Paolo Viotti i Marko Vukolić, *Consistency in Non-Transactional Distributed Storage Systems*, arXiv:1512.00168, 12 kwietnia 2016 (<https://arxiv.org/abs/1512.00168>).
- [6] Maurice P. Herlihy i Jeannette M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, „ACM Transactions on Programming Languages and Systems” (TOPLAS), rocznik 12, nr 3, s. 463 – 492, lipiec 1990 (<http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf>; <https://dl.acm.org/citation.cfm?doid=78969.78972>).
- [7] Leslie Lamport, *On Interprocess Communication*, „Distributed Computing”, rocznik 1, nr 2, s. 77 – 101, czerwiec 1986 (<https://www.microsoft.com/en-us/research/publication/interprocess-communication-part-basic-formalism-part-ii-algorithms/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Finterprocess.pdf>; <https://link.springer.com/article/10.1007%2FBF01786228>).
- [8] David K. Gifford, *Information Storage in a Decentralized Computer System*, Xerox Palo Alto Research Centers, CSL-81-8, czerwiec 1981 ([http://www.mirror-service.org/sites/www.bitsavers.org/pdf/xerox/parc/techReports/CSL-81-8\\_Information\\_Storage\\_in\\_a\\_Decentralized\\_Computer\\_System.pdf](http://www.mirror-service.org/sites/www.bitsavers.org/pdf/xerox/parc/techReports/CSL-81-8_Information_Storage_in_a_Decentralized_Computer_System.pdf)).
- [9] Martin Kleppmann, *Please Stop Calling Databases CP or AP*, [martin.kleppmann.com](http://martin.kleppmann.com), 11 maja 2015 (<http://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>).
- [10] Kyle Kingsbury, *Call Me Maybe: MongoDB Stale Reads*, [aphyr.com](http://aphyr.com), 20 kwietnia 2015 (<https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads>).
- [11] Kyle Kingsbury, *Computational Techniques in Knossos*, [aphyr.com](http://aphyr.com), 17 maja 2014 (<https://aphyr.com/posts/314-computational-techniques-in-knossos>).
- [12] Peter Bailis, *Linearizability Versus Serializability*, [bailis.org](http://bailis.org), 24 września 2014 (<http://www.bailis.org/blog/linearizability-versus-serializability/>).
- [13] Philip A. Bernstein, Vassos Hadzilacos i Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987, ISBN: 978-0-201-10715-9 (<https://www.microsoft.com/en-us/research/people/philbe/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fpeople%2Fphilbe%2Fcontrol.aspx>).



- [14] Mike Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, w: „7th USENIX Symposium on Operating System Design and Implementation” (OSDI), listopad 2006 (<https://research.google.com/archive/chubby.html>).
- [15] Flavio P. Junqueira i Benjamin Reed, *ZooKeeper: Distributed Process Coordination*, O’Reilly Media, 2013, ISBN: 978-1-449-36130-3.
- [16] *etcd 2.0.12 Documentation*, CoreOS, Inc., 2015.
- [17] *Apache Curator*, Apache Software Foundation, [curator.apache.org](http://curator.apache.org), 2015 (<http://curator.apache.org/>).
- [18] Morali Vallath, *Oracle 10g RAC Grid, Services & Clustering*, Elsevier Digital Press, 2006, ISBN: 978-1-555-58321-7.
- [19] Peter Bailis, Alan Fekete, Michael J Franklin i in., *Coordination-Avoiding Database Systems*, „Proceedings of the VLDB Endowment”, rocznik 8, nr 3, s. 185 – 196, listopad 2014 (<https://arxiv.org/pdf/1402.2237.pdf>).
- [20] Kyle Kingsbury, *Call Me Maybe: etcd and Consul*, [aphyr.com](http://aphyr.com), 9 czerwca 2014 (<https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul>).
- [21] Flavio P. Junqueira, Benjamin C. Reed i Marco Serafini, *Zab: HighPerformance Broadcast for Primary-Backup Systems*, w: „41st IEEE International Conference on Dependable Systems and Networks” (DSN), czerwiec 2011 (<https://pdfs.semanticscholar.org/b02c/6b00bd5dbdbd951fddb00b906c82fa80f0b3.pdf>; <http://ieeexplore.ieee.org/document/5958223/>).
- [22] Diego Ongaro i John K. Ousterhout, *In Search of an Understandable Consensus Algorithm (Extended Version)*, w: „USENIX Annual Technical Conference” (ATC), czerwiec 2014 (<http://ramcloud.stanford.edu/raft.pdf>).
- [23] Hagit Attiya, Amotz Bar-Noy i Danny Dolev, *Sharing Memory Robustly in Message-Passing Systems*, „Journal of the ACM”, rocznik 42, nr 1, s. 124 – 142, styczeń 1995 (<http://www.cse.huji.ac.il/course/2004/dist/p124-attiya.pdf>; <https://dl.acm.org/citation.cfm?doid=200836.200869>).
- [24] Nancy Lynch i Alex Shvartsman, *Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts*, w: „27th Annual International Symposium on Fault-Tolerant Computing” (FTCS), czerwiec 1997 (<http://groups.csail.mit.edu/tds/papers/Lynch/FTCS97.pdf>; <http://ieeexplore.ieee.org/document/614100/>).
- [25] Christian Cachin, Rachid Guerraoui i Luís Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, wydanie drugie, Springer, 2011, ISBN: 978-3-642-15259-7 (<http://www.distributedprogramming.net/>; <https://link.springer.com/book/10.1007%2F978-3-642-15260-3>).
- [26] Sam Elliott, Mark Allen i Martin Kleppmann, wymiana informacji w wątku w serwisie [twitter.com](https://twitter.com), 15 października 2015 (<https://twitter.com/lenary/status/654761711933648896>).
- [27] Niklas Ekström, Mikhail Panchenko i Jonathan Ellis, *Possible Issue with Read Repair?*, wątek na liście dyskusyjnej *cassandra-dev*, październik 2012 ([http://mail-archives.apache.org/mod\\_mbox/cassandra-dev/201210.mbox/%3CFA480D1DC3964E2C8B0A14E0880094C9%40Robotech%3E](http://mail-archives.apache.org/mod_mbox/cassandra-dev/201210.mbox/%3CFA480D1DC3964E2C8B0A14E0880094C9%40Robotech%3E)).

- [28] Maurice P. Herlihy, *Wait-Free Synchronization*, „ACM Transactions on Programming Languages and Systems” (TOPLAS), rocznik 13, nr 1, s. 124 – 149, styczeń 1991 (<https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf>; <https://dl.acm.org/citation.cfm?doid=114005.102808>).
- [29] Armando Fox i Eric A. Brewer, *Harvest, Yield, and Scalable Tolerant Systems*, w: „7th Workshop on Hot Topics in Operating Systems” (HotOS), marzec 1999 (<https://radlab.cs.berkeley.edu/people/fox/static/pubs/pdfs/c18.pdf>; <http://ieeexplore.ieee.org/document/798396/>).
- [30] Seth Gilbert i Nancy Lynch, *Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, „ACM SIGACT News”, rocznik 33, nr 2, s. 51 – 59, czerwiec 2002 (<http://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf>; <https://dl.acm.org/citation.cfm?doid=564585.564601>).
- [31] Seth Gilbert i Nancy Lynch, *Perspectives on the CAP Theorem*, „IEEE Computer Magazine”, rocznik 45, nr 2, s. 30 – 36, luty 2012 (<http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf>; <http://ieeexplore.ieee.org/document/6122006/>).
- [32] Eric A. Brewer, *CAP Twelve Years Later: How the „Rules” Have Changed*, „IEEE Computer Magazine”, rocznik 45, nr 2, s. 23 – 29, luty 2012 (<http://cs609.cs.ua.edu/CAP12.pdf>; <http://ieeexplore.ieee.org/document/6133253/>).
- [33] Susan B. Davidson, Hector Garcia-Molina i Dale Skeen, *Consistency in Partitioned Networks*, „ACM Computing Surveys”, rocznik 17, nr 3, s. 341 – 370, wrzesień 1985 ([http://delab.csd.auth.gr/~dimitris/courses/mpc\\_fall05/papers/invalidation/acm\\_csur85\\_partitioned\\_network\\_consistency.pdf](http://delab.csd.auth.gr/~dimitris/courses/mpc_fall05/papers/invalidation/acm_csur85_partitioned_network_consistency.pdf); <https://dl.acm.org/citation.cfm?doid=5505.5508>).
- [34] Paul R. Johnson i Robert H. Thomas, *RFC 677: The Maintenance of Duplicate Databases*, „Network Working Group”, 27 stycznia 1975 (<https://tools.ietf.org/html/rfc677>).
- [35] Bruce G. Lindsay, Patricia Griffiths Selinger, Cesare Galtieri i in., *Notes on Distributed Databases*, IBM Research, raport badawczy RJ2571(33471), lipiec 1979 ([http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/\\$File/RJ2571.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf)).
- [36] Michael J. Fischer i Alan Michael, *Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network*, w: „1st ACM Symposium on Principles of Database Systems” (PODS), marzec 1982 (<http://www.cs.ucsb.edu/~agrawal/spring2011/ugrad/p70-fischer.pdf>; <https://dl.acm.org/citation.cfm?doid=588111.588124>).
- [37] Eric A. Brewer, *NoSQL: Past, Present, Future*, w: „QCon San Francisco”, listopad 2012 (<https://www.infoq.com/presentations/NoSQL-History>).
- [38] Henry Robinson, *CAP Confusion: Problems with „Partition Tolerance”*, *blog.cloudera.com*, 26 kwietnia 2010 (<http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/>).
- [39] Adrian Cockcroft, *Migrating to Microservices*, w: „QCon London”, marzec 2014 (<https://www.infoq.com/presentations/migration-cloud-native>).
- [40] Martin Kleppmann, *A Critique of the CAP Theorem*, arXiv:1509.05393, 17 września 2015 (<https://arxiv.org/abs/1509.05393>).

- [41] Nancy A. Lynch, *A Hundred Impossibility Proofs for Distributed Computing*, w: „8th ACM Symposium on Principles of Distributed Computing” (PODC), sierpień 1989 (<http://groups.csail.mit.edu/tds/papers/Lynch/podc89.pdf>; <https://dl.acm.org/citation.cfm?doid=72981.72982>).
- [42] Hagit Attiya, Faith Ellen i Adam Morrison, *Limitations of Highly-Available Eventually-Consistent Data Stores*, w: „ACM Symposium on Principles of Distributed Computing” (PODC), lipiec 2015 (<http://www.cs.technion.ac.il/people/mad/online-publications/podc2015-replds.pdf>; <https://dl.acm.org/citation.cfm?doid=2767386.2767419>).
- [43] Peter Sewell, Susmit Sarkar, Scott Owens i in., *x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors*, „Communications of the ACM”, rocznik 53, nr 7, s. 89 – 97, lipiec 2010 (<http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf>; <https://dl.acm.org/citation.cfm?doid=1785414.1785443>).
- [44] Martin Thompson, *Memory Barriers/Fences*, [mechanicalsympathy.blogspot.co.uk](http://mechanical-sympathy.blogspot.co.uk), 24 lipca 2011 (<https://mechanical-sympathy.blogspot.co.uk/2011/07/memory-barriersfences.html>).
- [45] Ulrich Drepper, *What Every Programmer Should Know About Memory*, [akkadia.org](http://akkadia.org), 21 listopada 2007 (<https://www.akkadia.org/drepper/cpumemory.pdf>).
- [46] Daniel J. Abadi, *Consistency Tradeoffs in Modern Distributed Database System Design*, „IEEE Computer Magazine”, rocznik 45, nr 2, s. 37 – 42, luty 2012 (<http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf>; <http://ieeexplore.ieee.org/document/6127847/>).
- [47] Hagit Attiya i Jennifer L. Welch, *Sequential Consistency Versus Linearizability*, „ACM Transactions on Computer Systems” (TOCS), rocznik 12, nr 2, s. 91 – 122, maj 1994 (<http://courses.csail.mit.edu/6.852/01/papers/p91-attiya.pdf>; <https://dl.acm.org/citation.cfm?doid=176575.176576>).
- [48] Mustaque Ahamad, Gil Neiger, James E. Burns i in., *Causal Memory: Definitions, Implementation, and Programming*, „Distributed Computing”, rocznik 9, nr 1, s. 37 – 49, marzec 1995 ([http://www-i2.informatik.rwth-aachen.de/i2/fileadmin/user\\_upload/documents/Seminar\\_MCMM11/Causal\\_memory\\_1996.pdf](http://www-i2.informatik.rwth-aachen.de/i2/fileadmin/user_upload/documents/Seminar_MCMM11/Causal_memory_1996.pdf); <https://link.springer.com/article/10.1007%2FBF01784241>).
- [49] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky i David G. Andersen, *Stronger Semantics for Low-Latency Geo-Replicated Storage*, w: „10th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), kwiecień 2013 (<https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf>).
- [50] Marek Zawirski, Annette Bieniusa, Valter Balegas i in., *SwiftCloud: FaultTolerant Geo-Replication Integrated All the Way to the Client Machine*, INRIA Research Report 8347, sierpień 2013 (<https://arxiv.org/abs/1310.3107>).
- [51] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein i Ion Stoica, *Bolt-on Causal Consistency*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2013 (<http://db.cs.berkeley.edu/papers/sigmod13-bolton.pdf>).

- [52] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar i in., *Challenges to Adopting Stronger Consistency at Scale*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf>).
- [53] Peter Bailis, *Causality Is Expensive (and What to Do About It)*, *bailis.org*, 5 lutego 2014 (<http://www.bailis.org/blog/causality-is-expensive-and-what-to-do-about-it/>).
- [54] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero i Victor Fonte, *Concise Server-Wide Causality Management for Eventually Consistent Data Stores*, w: „15th IFIP International Conference on Distributed Applications and Interoperable Systems” (DAIS), czerwiec 2015 ([http://haslab.uminho.pt/tome/files/global\\_logical\\_clocks.pdf](http://haslab.uminho.pt/tome/files/global_logical_clocks.pdf); [https://link.springer.com/chapter/10.1007%2F978-3-319-19129-4\\_6](https://link.springer.com/chapter/10.1007%2F978-3-319-19129-4_6)).
- [55] Rob Conery, *A Better ID Generator for PostgreSQL*, *rob.conery.io*, 29 maja 2014 (<https://rob.conery.io/2014/05/28/a-better-id-generator-for-postgresql/>).
- [56] Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, „Communications of the ACM”, rocznik 21, nr 7, s. 558 – 565, lipiec 1978 (<https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Ftime-clocks.pdf>; <https://dl.acm.org/citation.cfm?doid=359545.359563>).
- [57] Xavier Défago, André Schiper i Péter Urbán, *Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey*, „ACM Computing Surveys”, rocznik 36, nr 4, s. 372 – 421, grudzień 2004 ([https://dspace.jaist.ac.jp/dspace/bitstream/10119/4883/1/defago\\_et\\_al.pdf](https://dspace.jaist.ac.jp/dspace/bitstream/10119/4883/1/defago_et_al.pdf); <https://dl.acm.org/citation.cfm?doid=1041680.1041682>).
- [58] Hagit Attiya i Jennifer Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, wydanie drugie, John Wiley & Sons, 2004, ISBN: 978-0-471-45324-6 (<http://onlinelibrary.wiley.com/book/10.1002/0471478210>).
- [59] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran i in., *CORFU: A Shared Log Design for Flash Clusters*, w: „9th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), kwiecień 2012 (<https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final30.pdf>).
- [60] Fred B. Schneider, *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*, „ACM Computing Surveys”, rocznik 22, nr 4, s. 299 – 319, grudzień 1990 (<http://www.cs.cornell.edu/fbs/publications/smsurvey.pdf>).
- [61] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng i in., *Calvin: Fast Distributed Transactions for Partitioned Database Systems*, w: „ACM International Conference on Management of Data” (SIGMOD), maj 2012 (<http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf>).
- [62] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber i in., *Tango: Distributed Data Structures over a Shared Log*, w: „24th ACM Symposium on Operating Systems Principles” (SOSP), listopad 2013 (<https://www.microsoft.com/en-us/research/publication/tango-distributed-data-structures-over-a-shared-log/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F199947%2Ftango.pdf>; <https://dl.acm.org/citation.cfm?doid=2517349.2522732>).

- [63] Robbert van Renesse i Fred B. Schneider, *Chain Replication for Supporting High Throughput and Availability*, w: „6th USENIX Symposium on Operating System Design and Implementation” (OSDI), grudzień 2004 ([http://static.usenix.org/legacy/events/osdi04/tech/full\\_papers/renesse/renesse.pdf](http://static.usenix.org/legacy/events/osdi04/tech/full_papers/renesse/renesse.pdf)).
- [64] Leslie Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, „IEEE Transactions on Computers”, rocznik 28, nr 9, s. 690 – 691, wrzesień 1979 (<https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/>; <http://ieeexplore.ieee.org/document/1675439/>).
- [65] Enis Söztutar, Devaraj Das i Carter Shanklin, *Apache HBase High Availability at the Next Level*, *hortonworks.com*, 22 stycznia 2015 (<https://hortonworks.com/blog/apache-hbase-high-availability-next-level/>).
- [66] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava i in., *PNUTS: Yahoo!’s Hosted Data Serving Platform*, w: „34th International Conference on Very Large Data Bases” (VLDB), sierpień 2008 (<https://people.mpi-sws.org/~druschel/courses/ds/papers/cooper-pnuts.pdf>; <https://dl.acm.org/citation.cfm?doid=1454159.1454167>).
- [67] Tushar Deepak Chandra i Sam Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, „Journal of the ACM”, rocznik 43, nr 2, s. 225 – 267, marzec 1996 (<http://courses.csail.mit.edu/6.852/08/papers/CT96-JACM.pdf>; <https://dl.acm.org/citation.cfm?doid=226643.226647>).
- [68] Michael J. Fischer, Nancy Lynch i Michael S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, „Journal of the ACM”, rocznik 32, nr 2, s. 374 – 382, kwiecień 1985 (<https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf>; <https://dl.acm.org/citation.cfm?doid=3149.214121>).
- [69] Michael Ben-Or, *Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols*, w: „2nd ACM Symposium on Principles of Distributed Computing” (PODC), sierpień 1983 (<https://dl.acm.org/citation.cfm?id=806707>).
- [70] Jim N. Gray i Leslie Lamport, *Consensus on Transaction Commit*, „ACM Transactions on Database Systems” (TODS), rocznik 31, nr 1, s. 133 – 160, marzec 2006 (<http://db.cs.berkeley.edu/cs286/papers/paxoscommit-tods2006.pdf>; <https://dl.acm.org/citation.cfm?doid=1132863.1132867>).
- [71] Rachid Guerraoui, *Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus*, w: „9th International Workshop on Distributed Algorithms” (WDAG), wrzesień 1995 (<https://pdfs.semanticscholar.org/5d06/489503b6f791aa56d2d7942359c2592e44b0.pdf>; <https://link.springer.com/chapter/10.1007%2FBFb0022140>).
- [72] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan i in., *All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications*, w: „11th USENIX Symposium on Operating Systems Design and Implementation” (OSDI), październik 2014 (<http://research.cs.wisc.edu/wind/Publications/alice-osdi14.pdf>).
- [73] Jim Gray, *The Transaction Concept: Virtues and Limitations*, w: „7th International Conference on Very Large Data Bases” (VLDB), wrzesień 1981 (<http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf>).

- [74] Hector Garcia-Molina i Kenneth Salem, *Sagas*, w: „ACM International Conference on Management of Data” (SIGMOD), maj 1987 (<http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>; <https://dl.acm.org/citation.cfm?doid=38713.38742>).
- [75] C. Mohan, Bruce G. Lindsay i Ron Obermarck, *Transaction Management in the R\* Distributed Database Management System*, „ACM Transactions on Database Systems”, rocznik 11, nr 4, s. 378 – 396, grudzień 1986 (<https://cs.brown.edu/courses/csci2270/archives/2012/papers/dtxn/p378-mohan.pdf>; <https://dl.acm.org/citation.cfm?doid=7239.7266>).
- [76] *Distributed Transaction Processing: The XA Specification*, X/Open Company Ltd., standard techniczny XO/CAE/91/300, grudzień 1991, ISBN: 978-1-872-63024-3 (<http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>).
- [77] Mike Spille, *XA Exposed, Part II*, *jroller.com*, 3 kwietnia 2004.
- [78] Ivan Silva Neto i Francisco Reverbel, *Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction*, w: „7th IEEE/ACIS International Conference on Computer and Information Science” (ICIS), maj 2008 (<https://www.ime.usp.br/~reverbel/papers/icis2008.pdf>; <http://ieeexplore.ieee.org/document/4529847/>).
- [79] James E. Johnson, David E. Langworthy, Leslie Lamport i Friedrich H. Vogt, *Formal Specification of a Web Services Protocol*, w: „1st International Workshop on Web Services and Formal Methods” (WS-FM), luty 2004 (<https://www.microsoft.com/en-us/research/publication/formal-specification-of-a-web-services-protocol/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fwsfm-web.pdf>; <http://linkinghub.elsevier.com/retrieve/pii/S1571066104051655>).
- [80] Dale Skeen, *Nonblocking Commit Protocols*, w: „ACM International Conference on Management of Data” (SIGMOD), kwiecień 1981 (<http://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/Ske81.pdf>; <https://dl.acm.org/citation.cfm?doid=582318.582339>).
- [81] Gregor Hohpe, *Your Coffee Shop Doesn't Use Two-Phase Commit*, „IEEE Software”, rocznik 22, nr 2, s. 64 – 66, marzec 2005 (<https://www.martinfowler.com/ieeeSoftware/coffeeShop.pdf>; <http://ieeexplore.ieee.org/document/1407829/>).
- [82] Pat Helland, *Life Beyond Distributed Transactions: An Apostate's Opinion*, w: „3rd Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2007 (<http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf>).
- [83] Jonathan Oliver, *My Beef with MSDTC and Two-Phase Commits*, *blog.jonathanoliver.com*, 4 kwietnia 2011 (<http://blog.jonathanoliver.com/my-beef-with-msdtc-and-two-phase-commits/>).
- [84] Oren Eini (Ahende Rahien), *The Fallacy of Distributed Transactions*, *ayende.com*, 17 lipca 2014 (<https://ayende.com/blog/167362/the-fallacy-of-distributed-transactions>).
- [85] Clemens Vasters, *Transactions in Windows Azure (with Service Bus) — An Email Discussion*, *vasters.com*, 30 lipca 2012 (<https://blogs.msdn.microsoft.com/clemensv/2012/07/30/transactions-in-windows-azure-with-service-bus-an-email-discussion/>).

- [86] *Understanding Transactionality in Azure*, NServiceBus Documentation, Particular Software, 2015 (<https://docs.particular.net/nservicebus/azure/understanding-transactionality-in-azure>).
- [87] Randy Wigginton, Ryan Lowe, Marcos Albe i Fernando Ipar, *Distributed Transactions in MySQL*, w: „MySQL Conference and Expo”, kwiecień 2013 ([https://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/XA\\_final.pdf](https://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/XA_final.pdf)).
- [88] Mike Spille, *XA Exposed, Part I*, jroller.com, 3 kwietnia 2004.
- [89] Ajmer Dhariwal, *Orphaned MSDTC Transactions (-2 spids)*, eraofdata.com, 12 grudnia 2008 (<http://www.eraofdata.com/sql-server/troubleshooting-sql-server/orphaned-msdtc-transactions-2-spids/>).
- [90] Paul Randal, *Real World Story of DBCC PAGE Saving the Day*, sqlskills.com, 19 czerwca 2013 (<https://www.sqlskills.com/blogs/paul/real-world-story-of-dbcc-page-saving-the-day/>).
- [91] *in-doubt xact resolution Server Configuration Option*, dokumentacja systemu SQL Server 2016, Microsoft, Inc., 2016 (<https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/in-doubt-xact-resolution-server-configuration-option>).
- [92] Cynthia Dwork, Nancy Lynch i Larry Stockmeyer, *Consensus in the Presence of Partial Synchrony*, „Journal of the ACM”, rocznik 35, nr 2, s. 288 – 323, kwiecień 1988 (<https://www.net.t-labs.tu-berlin.de/~petr/ADC-07/papers/DLS88.pdf>; <https://dl.acm.org/citation.cfm?doid=42282.42283>).
- [93] Miguel Castro i Barbara H. Liskov, *Practical Byzantine Fault Tolerance and Proactive Recovery*, „ACM Transactions on Computer Systems”, rocznik 20, nr 4, s. 396 – 461, listopad 2002 (<http://zoo.cs.yale.edu/classes/cs426/2012/bib/castro02practical.pdf>; <https://dl.acm.org/citation.cfm?doid=571637.571640>).
- [94] Brian M. Oki i Barbara H. Liskov, *Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems*, w: „7th ACM Symposium on Principles of Distributed Computing” (PODC), sierpień 1988 (<http://www.cs.princeton.edu/courses/archive/fall11/cos518/papers/viewstamped.pdf>; <https://dl.acm.org/citation.cfm?doid=62546.62549>).
- [95] Barbara H. Liskov i James Cowling, *Viewstamped Replication Revisited*, Massachusetts Institute of Technology, raport techniczny MIT-CSAIL-TR-2012-021, lipiec 2012 (<http://pmg.csail.mit.edu/papers/vr-revisited.pdf>).
- [96] Leslie Lamport, *The Part-Time Parliament*, „ACM Transactions on Computer Systems”, rocznik 16, nr 2, s. 133 – 169, maj 1998 (<https://www.microsoft.com/en-us/research/publication/part-time-parliament/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Flamport-paxos.pdf>; <https://dl.acm.org/citation.cfm?doid=279227.279229>).
- [97] Leslie Lamport, *Paxos Made Simple*, „ACM SIGACT News”, rocznik 32, nr 4, s. 51 – 58, grudzień 2001 (<https://www.microsoft.com/en-us/research/publication/paxos-made-simple/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fpaxos-simple.pdf>).

- [98] Tushar Deepak Chandra, Robert Griesemer i Joshua Redstone, *Paxos Made Live — An Engineering Perspective*, w: „26th ACM Symposium on Principles of Distributed Computing” (PODC), czerwiec 2007 (<http://www.read.seas.harvard.edu/~kohler/class/08w-dsi/chandra07paxos.pdf>).
- [99] Robbert van Renesse, *Paxos Made Moderately Complex*, [cs.cornell.edu](http://www.cs.cornell.edu/home/rvr/Paxos/paxos.pdf), marzec 2011 (<http://www.cs.cornell.edu/home/rvr/Paxos/paxos.pdf>).
- [100] Diego Ongaro, *Consensus: Bridging Theory and Practice*, praca doktorska, Uniwersytet Stanforda, sierpień 2014 (<https://github.com/ongardie/dissertation>).
- [101] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy i Jon Crowcroft, *Raft Refloated: Do We Have Consensus?*, „ACM SIGOPS Operating Systems Review”, rocznik 49, nr 1, s. 12 – 21, styczeń 2015 (<http://www.cl.cam.ac.uk/~ms705/pub/papers/2015-osr-raft.pdf>; <https://dl.acm.org/citation.cfm?doid=2723872.2723876>).
- [102] André Medeiros, *ZooKeeper’s Atomic Broadcast Protocol: Theory and Practice*, Aalto University School of Science, 20 marca 2012 (<http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf>).
- [103] Robbert van Renesse, Nicolas Schiper i Fred B. Schneider, *Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab*, „IEEE Transactions on Dependable and Secure Computing”, rocznik 12, nr 4, s. 472 – 484, wrzesień 2014 (<https://arxiv.org/abs/1309.5671>; <http://ieeexplore.ieee.org/document/6894199/>).
- [104] Will Portnoy, *Lessons Learned from Implementing Paxos*, [blog.willportnoy.com](http://blog.willportnoy.com), 14 czerwca 2012 (<http://blog.willportnoy.com/2012/06/lessons-learned-from-paxos.html>).
- [105] Heidi Howard, Dahlia Malkhi i Alexander Spiegelman, *Flexible Paxos: Quorum Intersection Revisited*, arXiv:1608.06696, 24 sierpnia 2016 (<https://arxiv.org/abs/1608.06696>).
- [106] Heidi Howard i Jon Crowcroft, *Coracle: Evaluating Consensus at the Internet Edge*, w: „Annual Conference of the ACM Special Interest Group on Data Communication” (SIGCOMM), sierpień 2015 (<http://www.sigcomm.org/sites/default/files/ccr/papers/2015/August/2829988-2790010.pdf>; <https://dl.acm.org/citation.cfm?doid=2829988.2790010>).
- [107] Kyle Kingsbury, *Call Me Maybe: Elasticsearch 1.5.0*, [aphyr.com](http://aphyr.com), 27 kwietnia 2015 (<https://aphyr.com/posts/323-call-me-maybe-elasticsearch-1-5-0>).
- [108] Ivan Kelly, *BookKeeper Tutorial*, [github.com](https://github.com/ivankelly/bookkeeper-tutorial), październik 2014 (<https://github.com/ivankelly/bookkeeper-tutorial>).
- [109] Camille Fournier, *Consensus Systems for the Skeptical Architect*, w: „Craft Conference”, Budapeszt, Węgry, kwiecień 2015 (<http://www.ustream.tv/recorded/61483409>).
- [110] Kenneth P. Birman, *A History of the Virtual Synchrony Replication Model*, w: „Replication: Theory and Practice”, Springer LNCS, nr 5959, rozdział 6, s. 91 – 120, 2010, ISBN: 978-3-642-11293-5 (<https://www.truststc.org/pubs/713.html>; [https://link.springer.com/chapter/10.1007%2F978-3-642-11294-2\\_6](https://link.springer.com/chapter/10.1007%2F978-3-642-11294-2_6)).





# MORZE DANYCH POCHODNYCH



## Dane pochodne

W częściach I i II tej książki zebrano wszystkie najważniejsze kwestie związane z bazami rozproszonymi: od układu danych na dysku aż po ograniczenia spójności w środowisku rozproszonym, gdy występują błędy. Jednak w tych analizach przyjmowano, że aplikacja używa tylko jednej bazy.

W praktyce systemy danych często są bardziej złożone. Duże aplikacje nieraz wymagają różnych sposobów dostępu do danych i ich przetwarzania, a nie istnieje jedna baza, która spełnia wszystkie takie wymagania. Dlatego aplikacje często używają połączenia kilku różnych baz, indeksów, pamięci podręcznych, systemów analitycznych itd. oraz mechanizmów do przenoszenia danych między miejscami.

W ostatniej części tej książki omówione są kwestie związane z integrowaniem wielu systemów danych (potencjalnie z różnymi modelami danych i zoptymalizowanych pod kątem innych wzorców dostępu) w jedną spójną architekturę aplikacji. Ten aspekt budowania systemów zostaje często przeoczony przez producentów utrzymujących, że ich produkt potrafi zaspokoić wszystkie potrzeby. W rzeczywistości integrowanie różnych systemów jest często jednym z najważniejszych zadań niezbędnych w niebanalnych aplikacjach.

## Systemy zapisu a systemy danych pochodnych

Na ogólnym poziomie systemy przechowujące i przetwarzające dane można pogrupować w dwie szerokie kategorie:

### *Systemy zapisu*

System zapisu, nazywany też *źródłem faktów*, przechowuje wiarygodną wersję danych. Gdy pojawiają się nowe dane (np. dane wejściowe od użytkownika), najpierw są zapisywane właśnie tu. Każdy fakt jest reprezentowany tylko raz (reprezentacja jest zwykle *znormalizowana*). Jeśli występuje rozbieżność między innym systemem a systemem zapisu, z definicji poprawna jest wartość z systemu zapisu.

### *Systemy danych pochodnych*

Informacje w systemie danych pochodnych są wynikiem pobrania istniejących danych z innego systemu i przekształcenia lub przetworzenia ich w jakiś sposób. Jeśli utracisz dane pochodne, możesz je odtworzyć na podstawie danych źródłowych. Klasycznym przykładem jest

pamięć podręczna. Dane mogą być udostępniane z pamięci podręcznej, jeśli są tam dostępne. Jeżeli jednak taka pamięć nie zawiera potrzebnych danych, można wykorzystać używaną bazę. Do tej kategorii należą wartości zdenormalizowane, indeksy i widoki zmaterializowane. W systemach rekomendacji propozycje są często generowane na podstawie dzienników użytkowania.

Technicznie rzecz biorąc, dane pochodne są *nadmiarowe* — w tym sensie, że powielają istniejące informacje. Jednak często okazują się niezbędne do uzyskania wysokiej wydajności zapytań wymagających odczytu. Zwykle są *zdenormalizowane*. Na podstawie jednego źródła można wygenerować kilka różnych zbiorów danych, co pozwala przyjrzeć się danym z paru punktów widzenia.

Nie we wszystkich systemach w architekturze występuje wyraźny podział na systemy zapisu i systemy danych pochodnych. Jest to jednak bardzo przydatne rozróżnienie, ponieważ jednoznacznie opisuje przepływ danych w systemie. Bezpośrednio określa, które części systemu są powiązane z poszczególnymi wejściami i wyjściami oraz jakie są zależności między danymi.

Większość baz danych, systemów składowania danych i języków zapytań nie jest z natury systemem zapisu lub systemem danych pochodnych. Baza danych to tylko narzędzie. To, jak będziesz jej używał, zależy od Ciebie. Rozróżnienie na systemy zapisu i systemy danych pochodnych zależy nie od narzędzia, ale od sposobu wykorzystania go w aplikacji.

Dzięki jednoznaczному określeniu, które dane są pochodne od poszczególnych innych danych, można jasno opisać skomplikowaną architekturę systemu. Ta kwestia będzie powracać przez resztę tej książki.

## Przegląd rozdziałów

Zaczynamy od rozdziału 10., gdzie przeanalizowano wsadowe systemy przepływu danych (takie jak MapReduce). Zobaczysz, że takie rozwiązania zapewniają dobre narzędzia i reguły do budowania systemów danych działających na dużą skalę. W rozdziale 11. te pomysły zostały zastosowane do strumieni danych pozwalających wykonywać operacje tego samego rodzaju z mniejszym opóźnieniem. Rozdział 12. kończy książkę i obejmuje przegląd pomysłów na wykorzystanie opisanych narzędzi do budowania niezawodnych, skalowalnych i łatwych w konserwacji aplikacji w przyszłości.





# Przetwarzanie wsadowe

*System nie może odnieść sukcesu, jeśli jedna osoba ma na niego zbyt duży wpływ. Gdy wstępny projekt jest gotowy i stosunkowo solidny, zaczynają się prawdziwe testy, kiedy ludzie mający różne punkty widzenia przystępują do własnych eksperymentów.*

— Donald Knuth

W pierwszych dwóch częściach książki sporo pisano o *żądaniach* i *zapytaniach* oraz powiązanych z nimi *odpowiedziach* i *wynikach*. Ten model przetwarzania jest stosowany w wielu współczesnych systemach danych. Pytasz o coś lub przesyłasz instrukcje, a po jakimś czasie system (miejmy nadzieję) zwraca odpowiedź. W ten sposób działają bazy danych, pamięci podręczne, indeksy wyszukiwania, serwery WWW i liczne inne systemy.

W tego rodzaju systemach *online* (może to być przeglądarka żądająca strony lub usługa wywołująca zdalny interfejs API) zwykle przyjmuje się, że żądanie jest uruchamiane przez ludzkiego użytkownika, który oczekuje na odpowiedź. Użytkownik nie powinien czekać zbyt długo, dlatego dużo uwagi poświęca się *czasowi odpowiedzi* takich systemów (zob. punkt „Opis wydajności”).

Internet i rosnąca liczba interfejsów API opartych na protokołach HTTP i REST sprawiły, że model interakcji żądanie-odpowiedź stał się tak powszechny, iż łatwo jest go uznać za standard. Należy jednak pamiętać, że to nie jedyny sposób budowania systemów i że inne podejścia też mają zalety. Wprowadźmy rozróżnienie na trzy typy systemów:

## *Usługi (systemy online)*

Usługa oczekuje na żądanie (lub instrukcję) od klienta. Po otrzymaniu żądania próbuje obsłużyć je jak najszybciej i odesłać odpowiedź. Czas odpowiedzi jest zwykle podstawową miarą wydajności usługi, a często bardzo istotna jest dostępność (jeśli klient nie może dotrzeć do usługi, prawdopodobnie otrzyma komunikat o błędzie).

## *Systemy przetwarzania wsadowego (systemy offline)*

System przetwarzania wsadowego przyjmuje dużą ilość danych wejściowych, uruchamia przetwarzające je *zadanie* i generuje dane wyjściowe. Wykonywanie zadań może trwać sporo czasu (od kilku minut do kilku dni), dlatego zwykle nie występuje użytkownik czekający na jego zakończenie. Zamiast tego często planuje się okresowe uruchamianie zadań wsadowych (np. raz dziennie). Główną miarą wydajności zadania wsadowego jest zazwyczaj *przepustowość* (czas potrzebny na przetworzenie wejściowego zbioru danych o określonej wielkości). Przetwarzanie wsadowe jest tematem tego rozdziału.

Przetwarzanie strumieniowe znajduje się między przetwarzaniem online a przetwarzaniem offline (wsadowym). Nazywane jest też przetwarzaniem *czasu prawie rzeczywistego* (ang. *near-real-time* lub *nearline*). Taki system (podobnie jak system przetwarzania wsadowego) przyjmuje dane wejściowe i generuje dane wyjściowe (zamiast odpowiadać na żądania). Jednak zadanie strumieniowe reaguje na zdarzenia wkrótce po ich wystąpieniu, natomiast zadanie wsadowe operuje na stałym zbiorze danych wejściowych. Ta różnica umożliwia systemom przetwarzania strumieniowego uzyskanie niższego opóźnienia niż w analogicznych systemach wsadowych. Ponieważ przetwarzanie strumieniowe jest oparte na przetwarzaniu wsadowym, zostało opisane w rozdziale 11.

W tym rozdziale zobaczysz, że przetwarzanie wsadowe jest ważnym narzędziem przy budowaniu niezawodnych, skalowalnych i łatwych w konserwacji aplikacji. Na przykład MapReduce, opublikowany w 2004 r. algorytm przetwarzania wsadowego [1], został (zapewne zbyt entuzjastycznie) nazwany „algorytmem, który zapewnia Google’owi wysoką skalowalność” [2]. Później został on zaimplementowany w różnych otwartych systemach danych takich jak Hadoop, CouchDB i MongoDB.

MapReduce to stosunkowo niskopoziomowy model programowania (w porównaniu z równoległymi systemami przetwarzania opracowanymi przed laty na potrzeby hurtowni danych [3, 4]). Okazał się jednak ważnym krokiem naprzód, jeśli chodzi o skalę przetwarzania możliwą z użyciem powszechnie dostępnego sprzętu. Choć znaczenie modelu MapReduce obecnie spada [5], nadal warto go poznać, ponieważ jasno pokazuje, dlaczego i w jaki sposób przetwarzanie wsadowe się przydaje.

W rzeczywistości przetwarzanie wsadowe jest bardzo dawnym sposobem przetwarzania. Na długo przed wymyśleniem programowalnych komputerów cyfrowych stosowano maszyny licząco-analityczne z kartami dziurkowanymi (np. maszyny Holleritha używane w amerykańskim spisie powszechnym w 1890 r.), w których implementowano częściowo mechaniczną postać przetwarzania wsadowego do obliczania zagregowanych statystyk na podstawie wielu danych wejściowych. Ponadto MapReduce jest niezwykle podobny do wprowadzonych przez IBM elektromechanicznych maszyn do sortowania kart powszechnie używanych do przetwarzania danych biznesowych w latach 40. i 50. [7]. Historia lubi się powtarzać.

W tym rozdziale opisano MapReduce oraz kilka innych algorytmów i platform przetwarzania wsadowego. Omówiono tu, jak są one używane w nowoczesnych systemach danych. Jednak najpierw przedstawiono przetwarzanie danych z użyciem standardowych narzędzi uniksowych. Nawet jeśli je znasz, warto przypomnieć filozofię Uniksa, ponieważ idee i wnioski związane z tym systemem znajdują odbicie w niejednorodnych rozproszonych systemach danych działających w dużej skali.

## Przetwarzanie wsadowe z użyciem narzędzi uniksowych

Zacznijmy od prostego przykładu. Załóżmy, że serwer WWW za każdym razem, gdy obsługuje żądanie, dołącza wiersz do pliku dziennika. Jeśli używany jest domyślny format dziennika dostępu serwera nginx, taki wiersz może wyglądać tak:



```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115
Safari/537.36"
```

Jest to jeden wiersz podzielony ze względu na czytelność na kilka części. Obejmuje on wiele informacji. Aby je zinterpretować, trzeba się zapoznać z definicją formatu dziennika:

```
$remote_addr - $remote_user [$time_local] "$request"
$status $body_bytes_sent "$http_referer" "$http_user_agent"
```

Tak więc ten wiersz dziennika oznacza, że 27 lutego 2015 r. o godzinie 17:55:11 czasu UTC serwer otrzymał żądanie pliku `/css/typography.css` od klienta o adresie IP 216.58.210.78. Ten użytkownik nie był uwierzytelniony, dlatego pole `$remote_user` to myślnik (-). Status odpowiedzi to 200 (żądanie zakończyło się powodzeniem), a odpowiedź zajęła 3377 bajtów. Użyta przeglądarka to Chrome 40, która wczytała plik, ponieważ jest używany na stronie o adresie URL `http://martin.kleppmann.com/`.

## Prosta analiza dziennika

Istnieją różne narzędzia, które potrafią pobierać takie pliki dziennika i generować czytelne raporty na temat ruchu w witrynie. Jednak w ramach ćwiczenia zbudujemy własny program za pomocą podstawowych narzędzi uniksowych. Założmy, że chcesz znaleźć pięć najpopularniejszych stron z witryny. Możesz to zrobić w powłoce uniksowej w następujący sposób<sup>1</sup>:

```
cat /var/log/nginx/access.log | ❶
awk '{print $7}' | ❷
sort | ❸
uniq -c | ❹
sort -r -n | ❺
head -n 5 ❻
```

❶ Wczytywanie pliku dziennika.

❷ Podział każdego wiersza na pola na podstawie odstępów i zwrócenie tylko siódmego pola z każdego wiersza. To pole zawiera żądany adres URL. W przykładowym wierszu ten adres to `/css/typography.css`.

❸ Alfabetyczne sortowanie (instrukcja `sort`) listy żądanych adresów URL. Jeśli któregoś adresu zażądano *n* razy, to po sortowaniu plik zawiera ten sam adres URL powtórzony *n* razy w serii.

❹ Polecenie `uniq` filtruje powtarzające się wiersze w danych wejściowych, sprawdzając, czy dwa przyległe wiersze są identyczne. Opcja `-c` nakazuje zwrócenie także wartości licznika. Dla każdego adresu URL określone jest, ile razy pojawił się on w danych wejściowych.

❺ Drugie wywołanie `sort` sortuje dane według liczby (`-n`) rozpoczynającej każdy wiersz (określa ona, ile razy zażądano danego adresu URL). Następnie wyniki są zwracane w odwrotnej kolejności (opcja `-r`), czyli od największej liczby do najmniejszej.

❻ W ostatnim kroku wywołanie `head` zwraca tylko pięć (`-n 5`) pierwszych wierszy z danych wejściowych i pomija pozostałe.

---

<sup>1</sup> Część osób z przyjemnością zwraca uwagę, że wywołanie `cat` jest tu zbędne, ponieważ plik wejściowy można bezpośrednio przekazać jako argument dla programu `awk`. Jednak potok liniowy jest bardziej czytelny, gdy go zapisać w ten sposób.

Dane wyjściowe z tej serii instrukcji wyglądają tak:

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
1369 /
915 /css/typography.css
```

Choć jeśli nie znasz narzędzi uniksowych, pokazana instrukcja wydaje się mało zrozumiała, jest ona bardzo wartościowa. Przetwarza gigabajtowe pliki dziennika w ciągu sekund, a analizy można łatwo dostosować do własnych potrzeb. Jeśli np. chcesz pominąć w raporcie pliki CSS, zmień argument wywołania `awk` na: `'$7 !~ /\.css$/ {print $7}'`; aby zliczać najczęściej występujące adresy IP klientów (zamiast najpopularniejszych stron), zmień argument wywołania `awk` na: `'{print $1}'` itd.

W tej książce nie ma miejsca na szczegółowe omawianie narzędzi uniksowych. Warto jednak się z nimi zapoznać. Zaskakująco wiele analiz danych można wykonać w kilka minut za pomocą kombinacji wywołań `awk`, `sed`, `grep`, `sort`, `uniq` i `xargs`. Ponadto ich wydajność jest niezwykle wysoka [8].

## Łącuch wywołań a niestandardowe programy

Zamiast stosować łańcuch wywołań uniksowych, możesz napisać prosty program wykonujący to samo zadanie. Na przykład w języku Ruby może on wyglądać tak:

```
counts = Hash.new(0) ❶

File.open('/var/log/nginx/access.log') do |file|
  file.each do |line|
    url = line.split[6] ❷
    counts[url] += 1 ❸
  end
end

top5 = counts.map{|url, count| [count, url] }.sort.reverse[0...5] ❹
top5.each{|count, url| puts "#{count} #{url}" } ❺
```

❶ Zmienna `counts` to tablica z haszowaniem przechowująca licznik wystąpień każdego adresu URL. Domyślnie wartość licznika to zero.

❷ Z każdego wiersza dziennika pobierany jest adres URL, czyli siódme pole uzyskane w wyniku podziału wiersza na podstawie odstępów (używany indeks tablicy to 6, ponieważ indeksy tablic w języku Ruby zaczynają się od zera).

❸ Zwiększanie licznika dla adresu URL z danego wiersza dziennika.

❹ Sortowanie zawartości tablicy z haszowaniem według wartości licznika (malejąco) i pobieranie pięciu pierwszych elementów.

❺ Wyświetlanie tych pierwszych pięciu elementów.

Ten program nie jest równie zwięzły jak łańcuch wywołań uniksowych, jednak jest stosunkowo czytelny. To, które rozwiązanie wybierzesz, pozostaje częściowo kwestią gustu. Jednak obok powierzchownych rozbieżności składniowych między tymi dwoma podejściami występuje tu istotna różnica w sposobie wykonywania kodu, która staje się oczywista, gdy uruchomisz te analizy na dużym pliku.

## Sortowanie a agregowanie w pamięci

Skrypt w języku Ruby przechowuje w pamięci tablicę z haszowaniem z adresami URL. W tej tablicy każdy adres URL jest powiązany z liczbą jego wystąpień. W przykładzie z potokiem wywołań uniksowych taka tablica nie jest używana. Zamiast tego sortowana jest lista adresów URL, na której powtarzające się adresy URL są duplikowane.

Które podejście jest lepsze? To zależy od liczby różnych adresów URL. W większości małych i średnich witryn zapewne wszystkie adresy URL oraz liczniki dla każdego z nich zmieszczą się w 1 GB pamięci. W tym przykładzie *zbiór roboczy* dla zadania (ilość potrzebnej zadaniu pamięci o dostępie swobodnym) zależy tylko od liczby różnych adresów URL. Jeśli w dzienniku znajduje się milion wpisów dla jednego adresu URL, w tablicy z haszowaniem i tak potrzebne jest miejsce tylko na jeden adres URL i licznik. Gdy zbiór roboczy jest odpowiednio mały, przechowywana w pamięci tablica z haszowaniem sprawdza się dobrze — nawet w laptopie.

Jeżeli jednak zbiór roboczy zadania jest większy niż ilość dostępnej pamięci, podejście z sortowaniem ma tę przewagę, że pozwala wydajnie wykorzystać dysk. Jest to ta sama zasada, którą opisano w punkcie „Pliki SSTable i drzewa LSM”. Porcje danych można sortować w pamięci i zapisywać na dysku jako pliki segmentu, a następnie wiele posortowanych segmentów można scalać w większy posortowany plik. W sortowaniu przez scalanie występują wzorce dostępu sekwencyjnego o wysokiej wydajności, gdy używany jest dysk. Pamiętaj, że optymalizacja pod kątem sekwencyjnych operacji wejścia-wyjścia była powtarzającym się tematem w rozdziale 3. Ta kwestia pojawia się i tutaj.

Narzędzie `sort` z linuksowego pakietu GNU Coreutils automatycznie obsługuje zbiory danych zajmujące więcej niż pojemność pamięci, zapisując je na dysku, i równolegle je sortuje z użyciem wielu rdzeni procesora [9]. To oznacza, że pokazany wcześniej prosty łańcuch poleceń uniksowych łatwiej się skaluje na potrzeby dużych zbiorów danych bez ryzyka wyczerpania się pamięci. Wąskim gardłem jest wtedy zwykle szybkość odczytu plików wejściowych z dysku.

## Filozofia Uniksa

To nie przypadek, że plik dziennika udało się stosunkowo łatwo przeanalizować za pomocą łańcucha poleceń takiego jak w poprzednim przykładzie. Było to jedno z najważniejszych założeń projektowych Uniksa, które do dziś pozostaje zaskakująco aktualne. Przyjrzyjmy się temu podejściu bliżej, aby zapożyczyć kilka pomysłów z Uniksa [10].

Doug McIlroy, pomysłodawca potoków uniksowych, po raz pierwszy opisał je w 1964 r. [11]: „Powinniśmy mieć sposoby łączenia programów tak, jak robimy to z węzłem ogrodowym — wprowadzając nowy segment, gdy konieczne się staje operowanie na danych w nowy sposób. To samo dotyczy operacji wejścia-wyjścia”. Analogia do hydrauliki się przyjęła, a pomysł łączenia programów w potoki stał się częścią czegoś, co obecnie nazywamy *filozofią Uniksa* — zestawu zasad projektowych, które zyskały popularność wśród programistów i użytkowników tego systemu. Ta filozofia tak została opisana w 1978 r. [12, 13]:

1. Spraw, by każdy program dobrze wykonywał jedną rzecz. W celu wykonania nowego zadania buduj rozwiązanie od nowa, zamiast komplikować starsze programy przez dodawanie nowych „funkcji”.

2. Zakładaj, że dane wyjściowe z każdego programu mogą się stać danymi wejściowymi następnego (na razie nieznanego) programu. Nie zaśmiecaj danych wyjściowych dodatkowymi informacjami. Unikaj ściśle kolumnowych lub binarnych formatów wejściowych. Nie wymuszaj interaktywnego podawania danych wejściowych.
3. Projektuj i buduj oprogramowanie (nawet systemy operacyjne) tak, by można je szybko wypróbować — najlepiej w ciągu kilku tygodni. Nie wahaj się usuwać niezgrabnych fragmentów i budować ich od nowa.
4. Stosuj narzędzia zamiast niewprawnej pomocy do ułatwienia wykonywania zadań programistycznych — nawet jeśli będziesz musiał poświęcić czas na zbudowanie tych narzędzi i wyrzucić niektóre z nich po zakończeniu ich używania.

To podejście (automatyzacja, szybkie tworzenie prototypów, przyrostowe iteracje, otwartość na eksperymenty i dzielenie dużych projektów na wykonalne fragmenty) jest zadziwiająco podobne do dzisiejszych ruchów Agile i DevOps. Przez 40 lat zmieniło się zdumiewająco niewiele.

Narzędzie `sort` to doskonały przykład programu, który wykonuje jedno zadanie dobrze. Zdaniem niektórych osób jest to lepsza implementacja sortowania niż w bibliotekach standardowych większości języków programowania (które nie zapisują danych na dysku i nie używają wielu wątków nawet w sytuacji, gdy jest to korzystne). Mimo to narzędzie `sort` samo w sobie jest mało przydatne. Staje się wartościowe tylko w połączeniu z innymi narzędziami uniksowymi takimi jak `uniq`.

Powłoka uniksowa (np. `bash`) umożliwia łatwe *łączenie* takich małych programów w zaskakująco wartościowe zadania przetwarzające dane. Choć wiele z tych programów zostało napisanych przez różne grupy ludzi, można je łączyć w elastyczny sposób. W jaki sposób Unix umożliwia to *łączenie*?

## Jednolity interfejs

Jeśli się spodziewasz, że dane wyjściowe z jednego programu będą danymi wejściowymi następnego, oznacza to, iż te programy muszą używać tego samego formatu danych, czyli zgodnego interfejsu. Jeżeli chcesz móc przekazać dane wyjściowe z *dowolnego* programu jako dane wejściowe do *dowolnego* innego programu, oznacza to, że wszystkie programy muszą używać tego samego interfejsu wejściowego i wyjściowego.

W Unikse tym interfejsem jest plik (a dokładniej: deskryptor pliku). Plik to nic więcej jak uporządkowana sekwencja bajtów. Ponieważ ten interfejs jest tak prosty, może posłużyć do reprezentowania wielu różnych rzeczy: pliku w systemie plików, kanału komunikacyjnego z innym procesem (gniazda uniksowego, strumieni `stdin` i `stdout`), sterownika urządzenia (np. `/dev/audio` lub `/dev/lp0`), gniazda połączenia TCP itd. Łatwo jest to przyjąć za coś normalnego, jednak to zdumiewające, że te wszystkie różne mechanizmy mogą korzystać z jednolitego interfejsu, co pozwala je łączyć ze sobą<sup>2</sup>.

---

<sup>2</sup> Innym przykładowym jednolitym interfejsem są adresy URL i protokół HTTP — podstawowe komponenty internetu. URL identyfikuje konkretną rzecz (zasób) w witrynie. W każdej innej witrynie można umieścić odsyłacz do dowolnego adresu URL. Użytkownik przeglądarki może dzięki temu płynnie przechodzić między witrynami za pomocą odsyłaczy, choć serwerami mogą zarządzać zupełnie niepowiązane organizacje. Ta zasada obecnie wydaje się oczywista, jednak była ważnym czynnikiem, dzięki któremu internet odniósł sukces. Wcześniejsze systemy nie były tak jednolite. Na przykład w epoce systemów BBS (ang. *bulletin board system*) każdy system miał własny numer telefonu i inną szybkość transmisji danych. Odnosił z jednego systemu BBS do innego miał postać numeru telefonicznego i ustawień modemu.

Zgodnie z konwencją wiele programów uniksowych (choć nie wszystkie) traktuje tę sekwencję bajtów jako tekst w formacie ASCII. Ten fakt wykorzystano w przykładowej analizie dziennika. Programy `awk`, `sort`, `uniq` i `head` traktują plik wejściowy jak listę rekordów rozdzielonych znakiem `\n` (jest to znak nowego wiersza; `0x0A` w ASCII). Wybór znaku `\n` jest arbitralny (zdaniem niektórych osób lepszym wyborem byłby separator rekordów z formatu ASCII (`0x1E`), ponieważ jest przeznaczony do opisywanego celu [14]). Ważne jest to, że wszystkie wymienione programy korzystają z tego samego separatora rekordów, co umożliwia ich współdziałanie.

Parsowanie każdego rekordu (np. wiersza danych wejściowych) jest bardziej skomplikowane. Narzędzia uniksowe zwykle rozbijają wiersze na pola na podstawie odstępów lub tabulacji. Jednak używane są też pliki CSV (z danymi rozdzielanymi przecinkami), dane rozdzielane znakiem potoku i inne kodowania. Nawet stosunkowo proste narzędzie takie jak `xargs` udostępnia kilka opcji wiersza poleceń do określania sposobu parsowania danych wejściowych.

Jednolity interfejs w postaci tekstu w formacie ASCII zwykle działa, jednak nie jest wyjątkowo wygodny. W przykładowej analizie dziennika do pobrania adresu URL użyto wyrażenia `{print $7}`, co nie jest czytelne. W idealnym rozwiązaniu to wyrażenie miałoby postać `{print $request_url}` lub podobną. Wróćmy jeszcze do tej kwestii.

Choć jednolity interfejs Uniksa nie jest idealny, nawet kilkadziesiąt lat później jest czymś wyjątkowym. Niewiele programów współdziała i łączy się ze sobą tak dobrze jak narzędzia uniksowe. Nie da się łatwo połączyć zawartości konta poczty elektronicznej i historii zakupów w sklepach internetowych za pomocą niestandardowego narzędzia analitycznego w arkuszu kalkulacyjnym, a następnie przesłać wyników do sieci społeczności lub wiki. Dziś programy, które współdziałają tak płynnie jak narzędzia uniksowe, są wyjątkiem, a nie normą.

Nawet bazy z *tym samym modelem danych* często nie umożliwiają łatwego przekazywania danych między nimi. Ten brak integracji prowadzi do „bałkanizacji” danych.

## Oddzielanie logiki od infrastruktury

Inna cechą charakterystyczną narzędzi uniksowych jest używanie standardowych strumieni wejścia (`stdin`) i wyjścia (`stdout`). Jeśli uruchamiasz program i nie zmieniasz ustawień, standardowe wejście pochodzi z klawiatury, a standardowe wyjście trafia na ekran. Możesz jednak pobierać dane wejściowe z pliku i/lub kierować dane wyjściowe do pliku. Potoki umożliwiają zastosowanie standardowego wyjścia z jednego procesu jako standardowego wejścia w innym procesie (z niewielkim buforem w pamięci i bez zapisu całego pośredniego strumienia danych na dysku).

Program w razie potrzeby może bezpośrednio wczytywać i zapisywać pliki, jednak podejście uniksowe działa najlepiej, gdy program nie musi się przejmować konkretnymi ścieżkami do plików i korzysta ze standardowego wejścia i wyjścia. Dzięki temu użytkownik powłoki może powiązać wejście i wyjście z dowolnym programem. Dla programu nie jest istotne, skąd pochodzą dane wejściowe i gdzie trafiają dane wyjściowe. Można powiedzieć, że jest to forma *luźnego powiązania i późnego wiązania* [15] lub *odwrócenia sterowania* [16]. Oddzielenie infrastruktury wejść i wyjść od logiki programu pozwala łatwiej łączyć proste narzędzia w większe systemy.

---

Użytkownik musiał się rozłączyć, wybrać numer innego systemu BBS, a następnie ręcznie znaleźć szukane informacje. Nie dało się utworzyć odsyłacza prowadzącego bezpośrednio do konkretnych informacji z innego systemu BBS.

Możesz nawet pisać własne programy i łączyć je z narzędziami udostępnianymi przez system operacyjny. Program musi tylko wczytywać wejście ze `stdin` i zapisywać wyjście do `stdout`. Może wtedy być częścią potoków przetwarzania danych. W przykładowej analizie dziennika możesz napisać narzędzie przekształcające łańcuchy znaków reprezentujące agenta na bardziej czytelne identyfikatory przeglądarek lub narzędzie zmieniające adresy IP na kody państw, a następnie dodać taki program do potoku. Dla programu sort nie jest istotne, czy komunikuje się z innym komponentem systemu operacyjnego, czy z narzędziem napisanym przez Ciebie.

Jednak możliwości `stdin` i `stdout` są ograniczone. Programy wymagające wielu wejść i wyjść są możliwe do napisania, ale pisanie ich sprawia trudności. Nie możesz też przekazać wyjścia programu do połączenia sieciowego [17, 18]<sup>3</sup>. Jeśli program bezpośrednio otwiera pliki do odczytu i zapisu, uruchamia inny program jako proces podrzędny lub otwiera połączenie sieciowe, za wejście i wyjście odpowiada sam program. Wejście i wyjście można konfigurować (np. za pomocą opcji wiersza poleceń), jednak elastyczność łączenia wejść i wyjść w powłoce staje się mniejsza.

## Przezroczystość i eksperymenty

Jednym z czynników sprawiających, że narzędzia uniksowe odniosły tak duży sukces, jest stosunkowa łatwość zrozumienia, co się w nich dzieje:

- Pliki wejściowe dla poleceń Uniksa są zwykle traktowane jako niemodyfikowalne. To oznacza, że możesz uruchamiać polecenia dowolnie często i wypróbować różne opcje wiersza poleceń, nie uszkadzając przy tym plików wejściowych.
- Możesz zakończyć potok w dowolnym punkcie, przekazać wyjście do programu `less` i sprawdzić, czy dane wyglądają tak, jak tego oczekiwałeś. Możliwość badania danych jest cenna w trakcie debugowania.
- Możesz zapisać wyjście z jednego etapu potoku do pliku i wykorzystać ten plik jako wejście w następnym etapie. Dzięki temu będziesz mógł uruchomić późniejszy etap bez ponownego wykonywania całego potoku.

Dlatego choć narzędzia uniksowe są dość proste w porównaniu z optymalizatorem zapytań z baz relacyjnych, wciąż są niezwykle przydatne — zwłaszcza w ramach eksperymentów. Jednak największym ograniczeniem narzędzi uniksowych jest to, że działają tylko na jednej maszynie. W tym kontekście pojawiają się narzędzia takie jak Hadoop.

## MapReduce i rozproszone systemy plików

MapReduce przypomina nieco narzędzia uniksowe, jednak rozproszone potencjalnie między tysiące maszyn. MapReduce (podobnie jak programy uniksowe) to narzędzie stosunkowo proste i działające przez atak siłowy, ale zaskakująco skuteczne. Jedna praca w modelu MapReduce jest porównywalna z jednym procesem Uniksa. Przyjmuje przynajmniej jedno dane wejściowe i generuje przynajmniej jedno dane wyjściowe.

---

<sup>3</sup> Wyjątkiem jest użycie odrębnego narzędzia takiego jak `netcat` lub `curl`. W Uniksie próbowano reprezentować wszystko za pomocą plików, jednak interfejs API gniazd BSD jest niezgodny z tą konwencją [17]. Systemy operacyjne pisane przez naukowców, *Plan 9* i *Inferno*, są bardziej spójne w korzystaniu z plików i reprezentują połączenie TCP jako plik w katalogu `/net/tcp` [18].

Podobnie jak w większości narzędzi uniksowych uruchomienie pracy w modelu MapReduce zwykle nie modyfikuje danych wejściowych i nie ma żadnych efektów ubocznych oprócz wygenerowania danych wyjściowych. Pliki wyjściowe są zapisywane raz w modelu sekwencyjnym (bez modyfikowania istniejących fragmentów pliku po ich zapisaniu).

Narzędzia uniksowe używają strumieni `stdin` i `stdout` jako wejścia i wyjścia, natomiast prace w modelu MapReduce wczytują i zapisują pliki w rozproszonym systemie plików. W implementacji modelu MapReduce w Hadoopie ten system plików to **HDFS** (ang. *Hadoop Distributed File System*). Jest to otwarta implementacja systemu **GFS** (ang. *Google File System*) [19].

Obok HDFS istnieją też inne rozproszone systemy plików, np. GlusterFS i **QFS** (ang. *Quantcast File System*) [20], obiektowe usługi składowania danych, np. Amazon S3, Azure Blob Storage i Open-Stack Swift [21], są pod wieloma względami podobne do nich<sup>4</sup>. W tym rozdziale w przykładach używany jest głównie HDFS, jednak opisane zasady obowiązują we wszystkich rozproszonych systemach plików.

HDFS działa *bez zasobów współdzielonych* (zob. wprowadzenie do części II), czym różni się od podejścia ze współużytkowanym dyskiem z architektur NAS (*Network Attached Storage*) i SAN (*Storage Area Network*). Pamięć z dyskiem współużytkowanym jest zwykle implementowana za pomocą centralnego urządzenia pamięci masowej, często z użyciem niestandardowego sprzętu i specjalnej infrastruktury sieciowej takiej jak magistrała Fibre Channel. Natomiast podejście bez zasobów współdzielonych nie wymaga specjalnego sprzętu, a tylko komputerów połączonych tradycyjną siecią w centrum danych.

HDFS obejmuje proces demona, który działa na każdej maszynie i udostępnia usługę sieciową pozwalającą innym węzłom na dostęp do plików z tej maszyny (przy założeniu, że w centrum danych każda maszyna do ogólnego użytku jest podłączona do dysków). Serwer centralny, *NameNode*, rejestruje, które bloki pliku są przechowywane w poszczególnych maszynach. Tak więc HDFS tworzy jeden duży system plików, który może wykorzystywać miejsce z dysków wszystkich maszyn, gdzie działa demon.

Aby zapewnić odporność na awarie maszyn i dysków, bloki pliku są replikowane w wielu maszynach. Replikacja może oznaczać wykonanie kilku kopii tych samych danych na wielu maszynach (tak jak w rozdziale 5.) lub stosowanie *kodowania nadmiarowego* (ang. *erasure coding*), np. kodowanie Reeda-Solomona, co umożliwia odzyskanie utraconych danych przy niższych kosztach pamięciowych niż w pełnej replikacji [20, 22]. Te techniki są podobne do macierzy RAID, które zapewniają nadmiarowość z użyciem kilku dysków podłączonych do tej samej maszyny. Różnica polega na tym, że w rozproszonym systemie plików dostęp do plików i replikacja odbywają się w tradycyjnej sieci centrum danych bez stosowania specjalnego sprzętu.

HDFS dobrze się skaluje. W czasie, gdy powstaje ta książka, największe instalacje z systemem HDFS obejmują dziesiątki tysięcy maszyn z łączną pamięcią setek petabajtów [23]. Osiągnięcie tak dużej skali jest możliwe, ponieważ w systemie HDFS koszt pamięci masowej i dostępu (dzięki

---

<sup>4</sup> Jedną z różnic polega na tym, że w systemie HDFS zadania obliczeniowe można uruchamiać w maszynie przechowującej kopię konkretnego pliku, natomiast obiektowe magazyny danych zwykle rozdzielają pamięć od obliczeń. Jeśli przepustowość sieci jest wąskim gardłem, odczyt z dysku lokalnego zapewnia lepszą wydajność. Zauważ jednak, że kodowanie nadmiarowe powoduje utratę korzyści płynących z lokalnego przetwarzania, ponieważ odtworzenie pierwotnego pliku wymaga połączenia danych z kilku maszyn [20].

standardowemu sprzętowi i otwartemu oprogramowaniu) jest znacznie niższy niż uzyskanie podobnych parametrów za pomocą specjalnie do tego przeznaczonych urządzeń pamięci masowej [24].

## Wykonywanie zadań w modelu MapReduce

MapReduce to platforma programowania umożliwiająca pisanie kodu do przetwarzania dużych zbiorów danych w rozproszonych systemach plików (takich jak HDFS). Najłatwiej zrozumieć to na przykładzie analizy dziennika z serwera WWW (zob. punkt „Prosta analiza dziennika”). Wzorzec przetwarzania danych w modelu MapReduce wygląda bardzo podobnie jak w tym przykładzie:

1. Wczytanie zestawu plików wejściowych i ich podział na *rekordy*. W przykładowym dzienniku serwera WWW każdy rekord to jeden wiersz z dziennika (separator rekordów jest więc znak `\n`).
2. Wywołanie mappera w celu pobrania klucza i wartości z każdego rekordu wejściowego. W omawianym przykładzie mapperem jest awk `'{print $7}'`. Pobiera on adres URL (`$7`) jako klucz i pozostawia pustą wartość.
3. Sortowanie wszystkich par klucz-wartość według klucza. W przykładzie odpowiada za to pierwsze wywołanie sort.
4. Wywołanie reducera na potrzeby iteracji po posortowanych parach klucz-wartość. Jeśli jest wiele wystąpień tego samego klucza, sortowanie sprawia, że znajdują się one obok siebie na liście. Dlatego można łatwo połączyć te wartości bez konieczności przechowywania w pamięci rozbudowanego stanu. W omawianym przykładzie reducer to polecenie `uniq -c`, które zlicza przyległe rekordy o tym samym kluczu.

Te cztery kroki można wykonać w jednej pracy w modelu MapReduce. Kroki 2. (mapowanie) i 4. (redukcja) wymagają napisania niestandardowego kodu do przetwarzania danych. Krok 1. (podział plików na rekordy) jest obsługiwany przez parser dla formatu wejściowego. Krok 3. (sortowanie) jest w modelu MapReduce wykonywany domyślnie. Nie musisz pisać kodu tego kroku, ponieważ wyjście z mappera jest zawsze sortowane przed przekazaniem go do reducera.

Aby utworzyć pracę w modelu MapReduce, musisz zaimplementować dwie wywoływane zwrrotnie funkcje (mapper i reducer), które działają w następujący sposób (zob. też punkt „Zapytania w modelu MapReduce”):

### *Mapper*

Jest wywoływany raz dla każdego rekordu wejściowego. Jego zadanie polega na pobraniu klucza i wartości z rekordu wejściowego. Dla każdego wejścia ta funkcja może wygenerować dowolną liczbę (także zero) par klucz-wartość. Nie przechowuje ona stanu między obsługą kolejnych rekordów, dlatego każdy rekord jest obsługiwany niezależnie.

### *Reducer*

Platforma MapReduce przyjmuje pary klucz-wartość wygenerowane przez mapper, łączy wszystkie wartości powiązane z tym samym kluczem i wywołuje reducera z użyciem iteratora dla kolekcji wartości. Reducer może generować rekordy wyjściowe (np. liczbę wystąpień tych samych adresów URL).



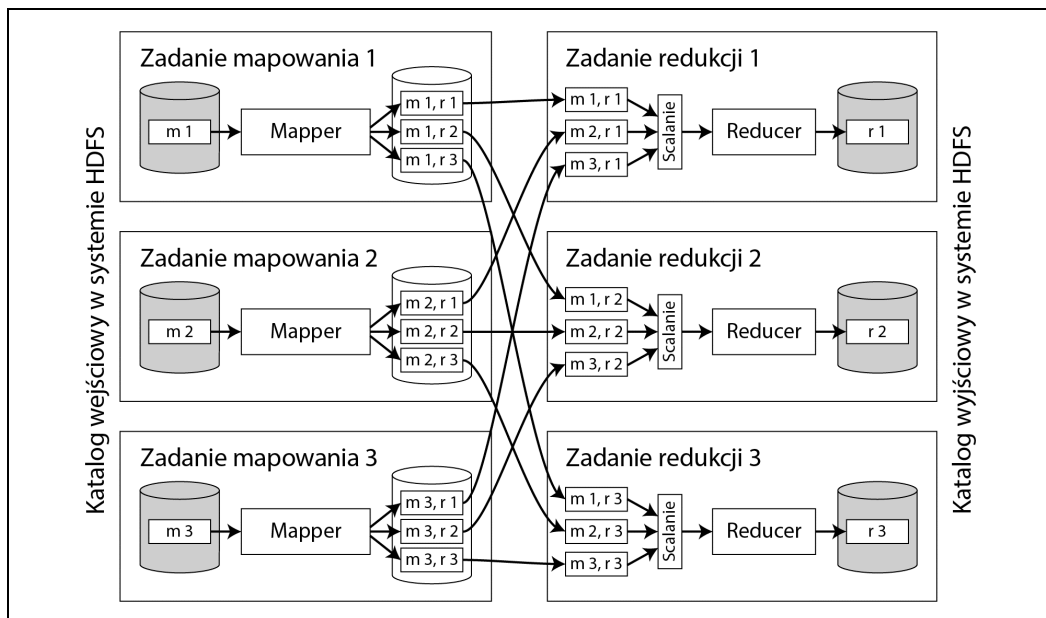
W przykładzie z dziennikiem serwera WWW w kroku 5. (zob. kod z punktu „Prosta analiza dziennika”) znajduje się drugie polecenie `sort`, sortujące adresy URL według liczby żądań. Jeśli w modelu MapReduce potrzebny jest drugi etap sortowania, można go zaimplementować za pomocą drugiej pracy i wykorzystać dane wyjściowe z pierwszej pracy jako dane wejściowe w drugiej. W tym podejściu rola mappera polega na przygotowaniu danych przez nadanie im formy odpowiedniej do sortowania, a reducer przetwarza posortowane dane.

## Rozproszone wykonywanie operacji w MapReduce

Główna różnica między potokami poleceń uniksowych a modelem MapReduce polega na tym, że MapReduce potrafi równolegle wykonywać obliczenia na wielu maszynach. Nie wymaga to pisania kodu do bezpośredniej obsługi równoległości. Mapper i reducer przetwarzają tylko jeden rekord naraz. Nie muszą wiedzieć, skąd pochodzą dane wejściowe i gdzie trafiają dane wyjściowe. To algorytm obsługuje złożone aspekty przenoszenia danych między maszynami.

Możliwe jest wykorzystanie standardowych narzędzi uniksowych jako mappera i reducera w obliczeniach rozproszonych [25]. Jednak częściej używane są funkcje napisane w tradycyjnym języku programowania. W modelu MapReduce w Hadoopie mapper i reducer to klasy Javy z implementacją określonego interfejsu. W MongoDB i CouchDB mapper i reducer to funkcje w JavaScriptcie (zob. punkt „Zapytania w modelu MapReduce”).

Na rysunku 10.1 pokazany jest przepływ danych w pracy w modelu MapReduce w Hadoopie. Równoległa praca jest oparta na partycjach (zob. rozdział 6.). Wejściem pracy jest zwykle katalog z systemu HDFS, a każdy plik lub blok pliku z katalogu wejściowego jest uznawany za odrębną partycję, którą można przetworzyć za pomocą osobnego zadania mapowania ( $m_1$ ,  $m_2$  i  $m_3$  na rysunku 10.1).



Rysunek 10.1. Praca w modelu MapReduce z trzema mapperami i trzema reducerami

Każdy plik wejściowy ma zwykle setki megabajtów wielkości. Program szeregujący w modelu MapReduce (niewidoczny na rysunku) próbuje uruchamiać każdy mapper w jednej z maszyn przechowujących replikę pliku wejściowego — pod warunkiem że maszyna ma wystarczającą ilość wolnej pamięci RAM i zasobów procesora, aby uruchomić zadanie mapowania [26]. Jest to zasada *umieszczania obliczeń blisko danych* [27]. Dzięki temu nie trzeba kopiować pliku wejściowego w sieci, co zmniejsza jej obciążenie i zwiększa lokalność.

W większości sytuacji kod aplikacji, który powinien działać w zadaniu mapowania, nie znajduje się w maszynie mającej go wykonywać. Dlatego platforma MapReduce najpierw kopiuje ten kod (np. pliki JAR, jeśli jest to program w Javie) na odpowiednie maszyny. Następnie uruchamia zadanie mapowania i rozpoczyna wczytywanie pliku wejściowego, przekazując po jednym rekordzie do wywołania zwrotnego mappera. Wyjście mappera to para klucz-wartość.

Także operacja redukcji jest rozdzielana. Liczba zadań mapowania zależy od liczby wejściowych bloków plików, natomiast liczba zadań redukcji jest konfigurowana przez autora pracy (może być ona inna niż liczba zadań mapowania). Aby mieć pewność, że wszystkie pary klucz-wartość o tym samym kluczu trafią do tego samego reducera, platforma używa skrótu klucza do określenia, które zadanie redukcji powinno odpowiadać za konkretną parę klucz-wartość (zob. punkt „Podział na podstawie skrótów kluczy”).

Pary klucz-wartość trzeba posortować, jednak zbiór danych jest zwykle zbyt duży, aby móc zastosować tradycyjny algorytm sortowania na jednej maszynie. Zamiast tego sortowanie jest wykonywane etapami. Najpierw każde zadanie mapowania rozdziela dane wyjściowe między reducery na podstawie skrótu klucza. Każda partycja jest zapisywana w posortowanym pliku na dysku lokalnym mappera za pomocą techniki podobnej do tej opisanej w punkcie „Pliki SSTable i drzewa LSM”.

Gdy mapper kończy wczytywanie pliku wejściowego i zapis posortowanych plików wyjściowych, program szeregujący w modelu MapReduce powiadamia reducery, że mogą zacząć pobierać pliki wyjściowe z mappera. Reducery łączą się z każdym z mapperów i pobierają pliki z posortowanymi parami klucz-wartość dla swojej partycji. Proces podziału danych według reducerów, sortowania i kopiowania partycji z mapperów do reducerów to *tasowanie* (ang. *shuffle*) [26] (jest to myląca nazwa, ponieważ w modelu MapReduce — inaczej niż w tasowaniu kart — nie jest to etap losowy).

Zadanie redukcji pobiera pliki z mapperów i scala je, zachowując porządek sortowania. Dlatego jeśli różne mappery generują rekordy o tym samym kluczu, w scalonym wejściu reducera rekordy te znajdują się obok siebie.

W wywołaniu reducera podawane są klucz i iterator, który po kolei skanuje wszystkie rekordy o tym samym kluczu (czasem nie wszystkie takie rekordy mieszczą się w pamięci). Reducer może przetwarzać rekordy za pomocą dowolnej logiki i wygenerować dowolną liczbę rekordów wyjściowych. Te rekordy wyjściowe są zapisywane w pliku w rozproszonym systemie plików (zwykle jedna kopia znajduje się na dysku lokalnym maszyny z danym reducerem, a na innych maszynach umieszczane są repliki).

## Przebieg pracy w modelu MapReduce

Zakres problemów, jakie można rozwiązać za pomocą jednej pracy w modelu MapReduce, jest ograniczony. Wróćmy do przykładowej analizy dziennika. Jedna praca w modelu MapReduce

może określić liczbę wyświetleń stron o poszczególnych adresach URL, ale już nie wskaże najpopularniejszych adresów URL, ponieważ wymaga to drugiego sortowania.

Dlatego prace w modelu MapReduce bardzo często są łączone w *przepływy pracy*, w których wyjście z jednej pracy staje się wejściem następnej. Model MapReduce w Hadoopie nie zapewnia specjalnej obsługi przepływów pracy, dlatego łańcuchy są tworzone pośrednio za pomocą nazw katalogów. Pierwsza praca musi być tak skonfigurowana, aby zapisywała dane wyjściowe do określonego katalogu w systemie HDFS. Druga praca ma wczytywać dane wejściowe z tego samego katalogu. W modelu MapReduce są to dwie niezależne prace.

Tak więc łańcuch zadań MapReduce mniej przypomina potoki poleceń uniksowych (przekazujących wyjście z jednego procesu bezpośrednio jako wejście następnego z użyciem niewielkiego bufora w pamięci), a bardziej sekwencję instrukcji, gdzie wyjście każdego polecenia jest zapisywane do pliku tymczasowego wczytywanego przez następne polecenie. Ten projekt ma wady i zalety opisane w punkcie „Materializowanie stanu pośredniego”.

Wyjście pracy wsadowej uznaje się za prawidłowe tylko wtedy, jeśli praca została poprawnie ukończona (MapReduce odrzuca częściowe wyjścia nieudanych prac). Dlatego praca w przepływie może rozpocząć działanie tylko wtedy, gdy wcześniejsze prace (generujące katalogi wejściowe dla danej) udanie się zakończyły. Aby poradzić sobie z zależnościami między wykonywanymi pracami, opracowano dla Hadoopa różne programy szeregujące przepływ pracy, takie jak Oozie, Azkaban, Luigi, Airflow i Pinball [28].

Te programy szeregujące udostępniają funkcje zarządzania przydatne do obsługi dużych zestawów zadań wsadowych. W systemach rekomendacji często zdarzają się przepływy pracy obejmujące od 50 do 100 zadań MapReduce [29], a w dużych organizacjach wiele różnych zespołów może uruchamiać prace wczytujące wyjścia z innych prac. Narzędzia są ważne do zarządzania takimi złożonymi przepływami danych.

Różne wysokopoziomowe narzędzia dla Hadoopa, np. Pig [30], Hive [31], Cascading [32], Crunch [33] i FlumeJava [34], także tworzą przepływy pracy z wieloma etapami zadań MapReduce automatycznie łączonymi w odpowiedni sposób.

## Złączenia i grupowanie na etapie redukcji

Złączenia zostały opisane w rozdziale 2. w kontekście modeli danych i języków zapytań. Nie wyjaśniono tam jednak, jak te złączenia są implementowane. Pora wrócić do tego wątku.

W wielu zbiorach danych często się zdarza, że jeden rekord jest powiązany z innym. Służą do tego *klucze obce* w modelu relacyjnym, *referencje do dokumentów* w modelu dokumentowym lub *krawędzie* w modelu opartym na grafach. Złączenie jest niezbędne, gdy kod wymaga dostępu do danych po obu stronach powiązania (zarówno do rekordu obejmującego referencję, jak i do docelowego rekordu). W rozdziale 2. opisano, że denormalizacja pozwala ograniczyć konieczność złączeń, ale zwykle nie eliminuje jej w pełni<sup>5</sup>.

---

<sup>5</sup> Złączenia omawiane w tej książce to zwykle *złączenia równościowe* (jest to najczęściej stosowany rodzaj złączeń), w których rekord jest wiązany z innymi rekordami o *identycznej wartości* w określonym polu (np. polu identyfikatora). Niektóre bazy obsługują inne rodzaje złączeń, np. z operatorem „mniejszy niż” zamiast operatora równości. Nie ma tu jednak miejsca na omawianie takich złączeń.

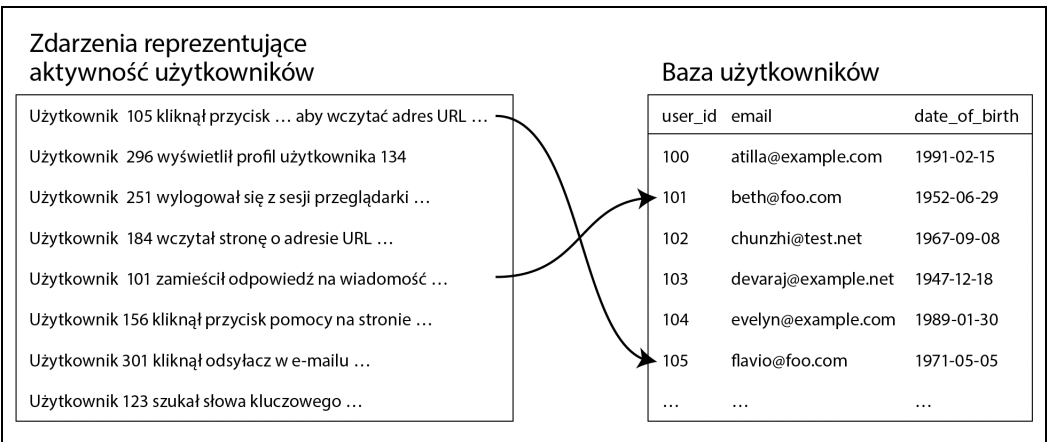
Jeśli wykonujesz zapytanie do bazy dotyczące niewielkiej liczby rekordów, baza zwykle używa *indeksu*, aby szybko znaleźć szukane dane (zob. rozdział 3.). Gdy zapytanie obejmuje złączenia, może wymagać wielokrotnego przeszukiwania indeksu. Jednak w MapReduce indeksy nie występują (przynajmniej nie w tradycyjnym sensie).

Gdy praca w modelu MapReduce otrzymuje na wejściu zestaw plików, wczytuje całą zawartość każdego z nich. W bazie taka operacja to *pełne skanowanie tabeli*. Jeśli chcesz wczytać tylko niewielką liczbę rekordów, pełne skanowanie tabeli jest nieakceptowalnie kosztowne w porównaniu z wyszukiwaniem na podstawie indeksu. Jednak w zapytaniach analitycznych (zob. punkt „Przetwarzanie transakcji czy analityka?”) często obliczane są agregacje na podstawie dużej liczby rekordów. Wtedy skanowanie całych danych wejściowych może być rozsądnym rozwiązaniem — zwłaszcza jeśli możesz przetwarzać dane równoległe z użyciem wielu maszyn.

Złączenia w przetwarzaniu wsadowym służą do znajdowania wszystkich wystąpień określonego powiązania w zbiorze danych. Zakładamy np., że praca jednocześnie przetwarza dane dotyczące wszystkich użytkowników, zamiast wyszukiwać dane jednej konkretnej osoby (co można zrobić znacznie wydajniej za pomocą indeksu).

### Przykład — analizy zdarzeń reprezentujących aktywność użytkownika

Typowy przykład złączenia w pracy wsadowej jest przedstawiony na rysunku 10.2. Po lewej stronie znajduje się dziennik zdarzeń opisujący, co zalogowani użytkownicy robili w witrynie (są to *zdarzenia reprezentujące aktywność* lub *dane o strumieniu kliknięć*). Po prawej widoczna jest baza użytkowników. Możesz przyjąć, że jest to część schematu gwiazdy (zob. punkt „Gwiazdy i płatki śniegu — schematy używane w analityce”). Dziennik zdarzeń to tabela faktów, a baza użytkowników to jeden z wymiarów.



Rysunek 10.2. Złączenie między dziennikiem zdarzeń reprezentujących aktywność użytkowników a bazą profili użytkowników

Zadanie analityczne może wymagać skorelowania aktywności użytkowników z informacjami z profilu. Na przykład jeśli profil obejmuje wiek lub datę urodzenia użytkowników, system może ustalić, które strony są najbardziej popularne wśród różnych grup wiekowych. Jednak zdarzenia związane

z aktywnością obejmują tylko identyfikator użytkownika, a nie pełne informacje z profilu. Umieszczanie informacji z profilu w każdym zdarzeniu byłoby marnotrawstwem. Dlatego zdarzenia reprezentujące aktywność trzeba złączyć z bazą profilu.

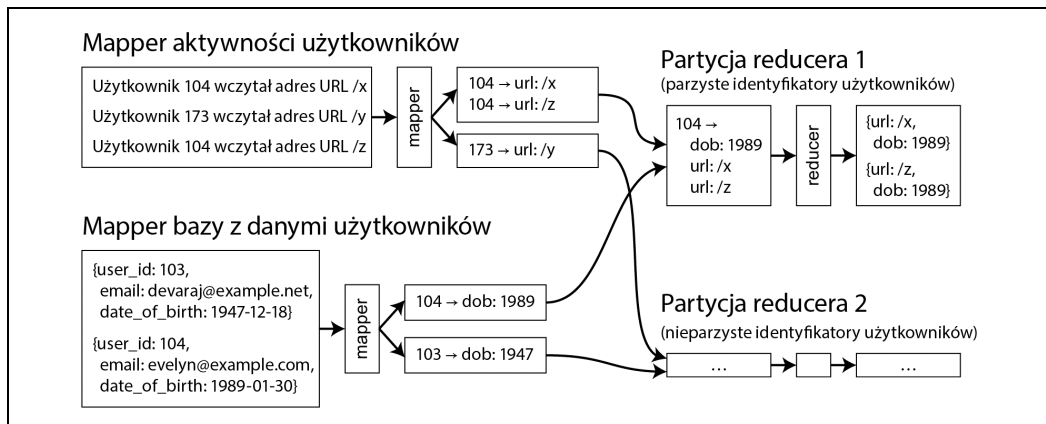
Najprostsza implementacja tego złączenia to pobieranie zdarzeń reprezentujących aktywność jedno po drugim i zgłaszanie zapytań do bazy użytkowników (ze zdalnego serwera) o każdy napotkany identyfikator. Jest to możliwe, jednak wydajność tego rozwiązania będzie prawdopodobnie bardzo niska. Przepustowość będzie ograniczona przez czas wymiany danych z serwerem bazodanowym, a skuteczność lokalnej pamięci podręcznej będzie w dużym stopniu zależna od rozkładu danych. Ponadto równoległe uruchamianie wielu zapytań może łatwo przeciążyć bazę [35].

Aby uzyskać wysoką przepustowość w procesie wsadowym, obliczenia trzeba w jak największym stopniu wykonywać lokalnie na jednej maszynie. Żądania z dostępem swobodnym przesyłane przez sieć dla każdego przetwarzanego rekordu są zbyt powolne. Ponadto zapytania do zdalnej bazy oznaczają, że praca wsadowa staje się niedeterministyczna (ponieważ dane w zdalnej bazie mogą się zmienić).

Dlatego lepszym podejściem jest skopiowanie bazy użytkowników (np. pobranej z kopii zapasowej za pomocą procesu ETL; zob. punkt „Hurtownie danych”) i umieszczenie jej w tym samym rozproszonym systemie plików, w którym znajduje się dziennik ze zdarzeniami reprezentującymi aktywność. Wtedy baza użytkowników znajduje się w jednym zestawie plików w systemie HDFS, a rekordy o aktywności — w innym zbiorze plików. MapReduce może wtedy łączyć wszystkie potrzebne rekordy w jednym miejscu i wydajnie je przetwarzać.

### Złączanie za pomocą sortowania przez scalanie

Przypomnij sobie, że celem mappera jest pobieranie klucza i wartości z każdego rekordu wejściowego. Na rysunku 10.2 kluczem jest identyfikator użytkownika. Jeden zestaw mapperów przetwarza zdarzenia reprezentujące aktywność (pobiera identyfikator użytkownika jako klucz i zdarzenie jako wartość), a inny operuje na bazie użytkowników (pobiera identyfikator użytkownika jako klucz i datę urodzenia tej osoby jako wartość). Ten proces jest pokazany na rysunku 10.3.



Rysunek 10.3. Złączanie za pomocą sortowania przez scalanie na podstawie identyfikatorów użytkownika po stronie reducera. Jeśli wejściowe zbiory danych są podzielone między wiele plików, każdy zbiór może być równoległe przetwarzany przez wiele mapperów

Gdy MapReduce dzieli dane wyjściowe z mappera na podstawie klucza, a następnie sortuje pary klucz-wartość, efekt jest taki, że wszystkie zdarzenia reprezentujące aktywność i rekordy użytkownika o tym samym identyfikatorze znajdują się obok siebie w danych wejściowych reducera. Praca w modelu MapReduce może nawet rozmieścić sortowane rekordy tak, by reducer zawsze najpierw natrafiał na rekordy z bazy użytkowników, a potem na zdarzenia uporządkowane według znaczników czasu. Ta technika to *sortowanie pomocnicze* [26].

Następnie reducer może łatwo złączyć dane. Funkcja reducera jest wywoływana raz dla każdego identyfikatora użytkownika, a dzięki sortowaniu pomocniczemu pierwszą oczekiwaną wartością jest rekord z datą urodzenia z bazy użytkowników. Reducer zapisuje datę urodzenia w zmiennej lokalnej, a następnie iteracyjnie pobiera zdarzenia reprezentujące aktywność mające ten sam identyfikator, aby zwracać pary *wyświetlony-url* plus *wiek-użytkownika-w-latach*. Dalsze prace w modelu MapReduce mogą obliczyć rozkład wieku użytkowników dla każdego adresu URL i pogrupować adresy na podstawie grup wiekowych.

Ponieważ reducer przetwarza wszystkie rekordy dla konkretnego identyfikatora w jednym kroku, w danym momencie musi przechowywać w pamięci tylko jeden rekord użytkownika. Nigdy nie musi też zgłaszać żądań przez sieć. Ten algorytm to *złączanie za pomocą sortowania przez scalanie*, ponieważ dane wyjściowe mappera są sortowane według klucza, a reducery scalają posortowane listy rekordów występujących po obu stronach złączenia.

### Łączenie powiązanych danych w jednym miejscu

W złączaniu za pomocą sortowania przez scalanie mappery i proces sortowania gwarantują, że wszystkie dane potrzebne w złączaniu dotyczącym konkretnego identyfikatora są dostępne w jednym miejscu — w jednym wywołaniu reducera. Dzięki wcześniejszemu przygotowaniu wszystkich potrzebnych danych reducer może być stosunkowo prostym, jednowątkowym fragmentem kodu, który potrafi przetwarzać rekordy z wysoką przepustowością i niskimi kosztami pamięciowymi.

Jeden ze sposobów patrzenia na tę architekturę jest taki, że mappery „przesyłają komunikaty” do reducerów. Gdy mapper generuje parę klucz-wartość, klucz działa jak adres docelowy, przykład, pod który należy dostarczyć wartość. Choć klucz jest dowolnym łańcuchem znaków (nie jest to rzeczywisty adres sieciowy taki jak IP i numer portu), działa jak adres. Wszystkie pary klucz-wartość o tym samym kluczu trafiają w to samo miejsce (do wywołania reducera).

Użycie modelu programowania MapReduce oddziela aspekty obliczeń związane z komunikacją w sieci fizycznej (przenoszenie danych na odpowiednią maszynę) od logiki aplikacji (przetwarzania danych, gdy są już dostępne). Ten podział różni się od typowych sposobów używania baz, gdzie żądanie pobrania danych z bazy często znajduje się gdzieś w kodzie aplikacji [36]. Ponieważ MapReduce obsługuje całą komunikację sieciową, sprawia też, że w kodzie aplikacji nie trzeba uwzględniać częściowych awarii (np. uszkodzenia innego węzła). MapReduce automatycznie ponawia nieudane prace bez wpływu na logikę aplikacji.

### GROUP BY

Obok złączeń innym częstym zastosowaniem wzorca „przenoszenie powiązanych danych w to samo miejsce” jest grupowanie rekordów na podstawie klucza (tak jak w klauzuli `GROUP BY` w `SQL-u`). Wszystkie rekordy o tym samym kluczu tworzą grupę, a następnym krokiem jest często przeprowadzanie agregacji w każdej grupie. Oto przykłady:

- Zliczanie rekordów w każdej grupie (tak jak w przykładzie zliczania odsłon strony, co w SQL-u można zapisać jako agregację `COUNT(*)`).
- Dodawanie wartości z określonego pola (`SUM(nazwa_pola)` w SQL-u).
- Wybór pierwszych  $k$  rekordów według określonej funkcji porządkującej.

Najprostszy sposób implementacji grupowania za pomocą modelu MapReduce polega na takim skonfigurowaniu mapperów, by generowane przez nie pary klucz-wartość miały odpowiedni klucz grupujący. Proces podziału i sortowania łączy wtedy wszystkie rekordy o tym samym kluczu w jednym reducerze. Dlatego grupowanie i złączenie implementowane za pomocą modelu MapReduce wyglądają podobnie.

Innym częstym zastosowaniem grupowania jest złączanie wszystkich zdarzeń reprezentujących aktywność z konkretnej sesji użytkownika w celu ustalenia sekwencji operacji wykonanych przez daną osobę. Ten proces to *podział na sesje* (ang. *sessionization*) [37]. Takie analizy można np. wykorzystać do ustalenia, czy użytkownicy, którym pokazano nową wersję witryny, dokonują zakupu częściej niż użytkownicy starszej wersji (są to testy A/B). W ten sposób można też ustalić, czy jakieś działania marketingowe przyniosły oczekiwany skutek.

Jeśli ządania użytkowników są obsługiwane przez wiele serwerów WWW, zdarzenia reprezentujące aktywność konkretnego użytkownika są zwykle rozproszone w plikach dziennika na różnych serwerach. Podział na sesje możesz zaimplementować za pomocą plików cookie sesji, identyfikatora użytkownika lub podobnego identyfikatora jako klucza grupującego. Pozwala to połączyć wszystkie zdarzenia związane z określonym użytkownikiem w jednym miejscu, a przy tym rozdzielić zdarzenia różnych użytkowników między partycje.

### Radzenie sobie z asymetrią

Wzorzec „przenoszenie wszystkich rekordów o tym samym kluczu w jedno miejsce” nie działa, jeśli bardzo duża ilość danych jest powiązana z jednym kluczem. Na przykład w sieci społecznościowej większość użytkowników jest połączona z kilkuset osobami, jednak niewielka liczba celebrytów może mieć miliony obserwatorów. Takie nieproporcjonalnie aktywne rekordy bazy można nazwać *obiektami osiowymi* (ang. *linchpin objects*) [38] lub *gorącymi kluczami* (ang. *hot keys*).

Rejestrowanie całej aktywności związanej z gwiazdą (np. odpowiedzi na zamieszczony przez nią wpis) w jednym reducerze może prowadzić do znacznej *asymetrii* (i powstawania *hot spotów*). Jeden reducer musi wtedy przetwarzać znacznie więcej rekordów niż pozostałe (zob. punkt „Asymetryczne obciążenie robocze i odciążanie hot spotów”). Ponieważ praca w modelu MapReduce jest ukończona tylko po zakończeniu działania przez wszystkie mappery i reducery, każda późniejsza praca musi oczekiwać na najwolniejszego reducera.

Jeśli dane wyjściowe używane w złączeniu obejmują „gorące klucze”, można skompensować to za pomocą kilku algorytmów. Na przykład w metodzie *złączania asymetrycznego* w narzędziu Pig najpierw uruchamiana jest praca próbkująca, aby ustalić, które klucze są „gorące” [39]. W trakcie samego złączania mappery przesyłają rekordy związane z „gorącym kluczem” do jednego z kilku losowo wybieranych reducerów (inaczej niż w tradycyjnym modelu MapReduce, gdzie reducery są wybierane deterministycznie na podstawie skrótu klucza). Aby możliwe było złączenie innych danych wejściowych, rekordy powiązane z „gorącym kluczem” trzeba zreplikować we *wszystkich* reducerach obsługujących dany klucz [40].

Ta technika rozdziela obsługę „gorących kluczy” między kilka reducerów. Dzięki temu operacja jest bardziej równoległa, ale kosztem konieczności replikowania innych danych wejściowych ze złączenia do wielu reducerów. Metoda *złączania z podziałem* (ang. *sharded join*) w bibliotece Crunch działa podobnie, ale wymaga bezpośredniego wskazania „gorących kluczy” zamiast stosowania pracy próbkującej. Ta technika bardzo przypomina rozwiązanie opisane w punkcie „Asymetryczne obciążenie robocze i odciążanie hot spotów”, ponieważ wykorzystuje losowość do odciążania hot spotów w bazie podzielonej na partycje.

Optymalizacja asymetrycznego złączania w narzędziu Hive wygląda inaczej. Wymaga bezpośredniego podania „gorących kluczy” w metadanych tabeli i zapisuje rekordy powiązane z takimi kluczami w odrębnych plikach. W trakcie złączania z użyciem tej tabeli dla „gorących kluczy” stosowane jest złączanie po stronie mappera.

W trakcie grupowania i agregowania rekordów na podstawie „gorących kluczy” grupowanie można przeprowadzić w dwóch etapach. Pierwszy etap modelu MapReduce przesyła rekordy do losowego reducera, tak aby każdy reducer przeprowadzał grupowanie podzbioru rekordów powiązanych z „gorącym kluczem” i zwracał zagregowaną wartość dla klucza. Później druga praca w modelu MapReduce łączy wartości z wszystkich reducerów z pierwszego etapu w jedną wartość dla klucza.

## Złączanie po stronie mappera

Algorytmy złączania opisane w poprzednim punkcie przeprowadzają złączanie logiczne w reducerach. Dlatego są nazywane *złączeniami po stronie reducera*. Mappery odpowiadają wtedy za przygotowywanie danych wejściowych: pobierają klucz i wartość z każdego rekordu wejściowego, przypisują pary klucz-wartość do partycji reducera i sortują dane na podstawie kluczy.

Złączenia po stronie reducera mają tę zaletę, że nie wymagają przyjmowania założeń co do danych wejściowych. Niezależnie od właściwości i struktury tych danych mappery mogą przygotować je do złączania. Wadą jest to, że sortowanie danych, ich kopiowanie do reducerów i scalanie danych wejściowych w reducerach mogą być kosztowne. W zależności od dostępnych buforów pamięci dane są czasem kilkakrotnie zapisywane na dysku, gdy przechodzą przez różne etapy modelu MapReduce [37].

Jeśli jednak *możesz* przyjąć określone założenia co do danych wejściowych, proces da się przyspieszyć dzięki *złączeniom po stronie mappera*. W tym podejściu wykorzystywana jest uproszczona praca w modelu MapReduce, w której nie ma ani reducerów, ani sortowania. Zamiast tego każdy mapper wczytuje jeden wejściowy blok pliku z rozproszonego systemu plików i zapisuje w tym systemie jeden plik wyjściowy — to wszystko.

## Złączanie z rozsyłaniem i haszowaniem

Najprostszy sposób złączania po stronie mappera można zastosować, gdy duży zbiór danych jest złączany z małym. Mały zbiór musi być na tyle niewielki, by można go było w całości umieścić w pamięci każdego mappera.

Wyobraź sobie, że baza użytkowników z rysunku 10.2 jest wystarczająco mała, iż mieści się w pamięci. Wtedy gdy mapper rozpoczyna pracę, najpierw może wczytać tę bazę z rozproszonego systemu plików do przechowywanej w pamięci tablicy z haszowaniem. Następnie mapper może ska-



nować zdarzenia reprezentujące aktywność użytkownika i szukać w tablicy identyfikatora użytkownika z każdego zdarzenia<sup>6</sup>.

Także wtedy da się wykonywać kilka zadań mapowania — po jednym dla każdego bloku pliku ze złączanymi długimi danymi wejściowymi (na rysunku 10.2 zdarzenia reprezentujące aktywność to duże dane wejściowe), a każdy mapper wczytuje do pamięci w całości małe dane wejściowe.

Ten prosty, ale skuteczny algorytm to *złączanie z rozsyłaniem i haszowaniem* (ang. *broadcast hash join*). Słowo *rozsyłanie* odzwierciedla tu to, że każdy mapper dla partycji dużych danych wejściowych wczytuje całe małe dane wejściowe (tak więc małe dane wejściowe są rozsyłane do wszystkich partycji dużych danych wejściowych). Słowo *haszowanie* dotyczy tu zastosowania tablicy z haszowaniem. Ta metoda złączania jest obsługiwana w narzędziach Pig (jako złączanie z replikacją), Hive (jako MapJoin), Cascading i Crunch. Jest też używana w hurtowniach danych w systemach obsługi zapytań takich jak Impala [41].

Zamiast wczytywać małe dane wejściowe do przechowywanej w pamięci tablicy z haszowaniem, można zapisać te dane w służącym tylko do odczytu indeksie na dysku lokalnym [42]. Często używane fragmenty tego indeksu będą się znajdować w pamięci podręcznej stron systemu operacyjnego, dlatego to podejście może zapewniać wyszukiwanie z dostępem swobodnym prawie tak szybkie jak tablica z haszowaniem w pamięci, ale nie wymaga tego, by zbiór danych mieścił się w pamięci.

## Złączanie z partycjami i haszowaniem

Jeśli dane wejściowe w złączaniu po stronie mappera są jednolicie podzielone na partycje, złączanie z haszowaniem można zastosować w każdej partycji z osobna. W przykładzie z rysunku 10.2 możesz podzielić zdarzenia reprezentujące aktywność i bazę użytkowników na partycje na podstawie ostatniej cyfry identyfikatora użytkownika (po każdej stronie występuje więc 10 partycji). Na przykład mapper 3 może wczytywać do tablicy z haszowaniem wszystkich użytkowników z identyfikatorem kończącym się cyfrą 3, a następnie skanować wszystkie zdarzenia, szukając tych związanych z użytkownikami z takimi identyfikatorami.

Jeżeli podział na partycje został przeprowadzony poprawnie, możesz mieć pewność, że wszystkie potrzebne rekordy znajdują się w partycji o tym samym numerze. Dlatego każdy mapper musi wczytać tylko jedną partycję każdego z wejściowych zbiorów danych. Ma to tę zaletę, że każdy mapper może wczytać do tablicy z haszowaniem mniej danych.

Ta technika działa tylko wtedy, jeśli oba wejścia w złączeniu mają tę samą liczbę partycji, a rekordy są przydzielane do partycji na podstawie tego samego klucza i tej samej funkcji haszującej. Jeśli dane wejściowe są generowane przez wcześniejsze prace w modelu MapReduce, które już przeprowadziły grupowanie, jest to sensowne założenie.

W narzędziu Hive [37] złączanie z partycjami i haszowaniem nosi nazwę *złączanie przez mapowanie z podziałem na kubeczki* (ang. *bucketed map join*).

---

<sup>6</sup> W tym przykładzie zakładamy, że w tablicy z haszowaniem dla każdego klucza występuje tylko jeden wpis, co w bazie użytkowników jest zapewne prawdą (identyfikator użytkownika jednoznacznie identyfikuje daną osobę). Zwykle jednak w tablicy z haszowaniem może się znajdować kilka wpisów dla jednego klucza, a operator złączania zwraca wszystkie dane pasujące dla określonego klucza.

## Złączanie przez scalanie po stronie mappera

Inny rodzaj złączania po stronie mappera można stosować, jeśli wejściowe zbiory danych nie tylko są podzielone w ten sam sposób, ale też *posortowane* według tego samego klucza. Tym samym nie jest ważne, czy dane wejściowe są wystarczająco małe, by zmieścić się w pamięci, ponieważ mapper może wykonać scalanie zwykle przeprowadzane przez reducer — może stopniowo wczytywać oba pliki wejściowe (uporządkowane rosnąco według kluczy) i dopasowywać do siebie rekordy mające ten sam klucz.

Gdy złączanie przez scalanie po stronie mappera jest możliwe, zwykle oznacza to, że wcześniejsze prace w modelu MapReduce podzieliły wejściowy zbiór danych na partycje i posortowały dane. Zasadniczo takie złączanie można przeprowadzić na etapie redukcji we wcześniejszej pracy. Jednak przydatne może być przeprowadzenie złączania przez scalanie w odrębnej pracy, obejmującej tylko mapowanie. Jest tak np. wtedy, gdy podzielone na partycje i posortowane zbiory danych są potrzebne także w innych celach oprócz złączania.

## Przepływy pracy w modelu MapReduce ze złączaniem po stronie mappera

Gdy dane wyjściowe ze złączania w modelu MapReduce są używane przez późniejsze prace, wybór między złączaniem po stronie mappera lub reducera wpływa na strukturę tych danych. Dane wyjściowe ze złączania po stronie reducera są dzielone i sortowane według klucza złączania. Natomiast dane wyjściowe ze złączania po stronie mappera są dzielone i sortowane w taki sam sposób, jak duże dane wyjściowe (ponieważ jedno zadanie mapowania jest uruchamiane dla każdego bloku pliku z dużymi danymi wejściowymi; jest tak niezależnie od tego, czy stosowane jest złączanie z partycjami, czy z rozsyłaniem).

Wcześniej opisano, że w złączaniu po stronie mappera można przyjmować założenia co do wielkości, posortowania i partycji wejściowych zbiorów danych. Znajomość fizycznego układu zbiorów danych w rozproszonym systemie plików staje się ważna w kontekście optymalizowania strategii złączania. Nie wystarczy wiedza o formacie kodowania i nazwie katalogu z danymi. Musisz też znać liczbę partycji i klucze, według których dane są dzielone i sortowane.

W środowisku Hadoopa tego rodzaju metadane związane z podziałem zbiorów danych są często przechowywane w systemie HCatalog i w repozytorium metadanych Hive'a [37].

## Dane wyjściowe ze wsadowych przepływów pracy

Opisano już różne algorytmy implementowania przepływów pracy w modelu MapReduce. Pominęto jednak ważne pytanie — jaki jest efekt tego przetwarzania? Po co w ogóle uruchamiane są te wszystkie prace?

W przypadku zapytań do bazy danych wprowadzono rozróżnienie na przetwarzanie transakcji (OLTP) i analitykę (zob. punkt „Przetwarzanie transakcji czy analityka?”). Zobaczyłeś, że zapytania OLTP zwykle wyszukują niewielką liczbę rekordów na podstawie klucza z użyciem indeksów, aby wyświetlić dane użytkownikowi (np. na stronie WWW). Natomiast zapytania analityczne zwykle skanują dużą liczbę rekordów, przeprowadzając grupowanie i agregację, a dane wyjściowe często mają postać raportu — wykresu pokazującego zmianę wartości wskaźnika w czasie, 10 pierwszych elementów według danego uporządkowania lub podział jakiejś wartości na podkategorie. Odbiorcą takiego raportu jest zazwyczaj analityk lub podejmujący decyzje biznesowe menedżer.

Gdzie wpasowuje się w to przetwarzanie wsadowe? Nie jest to ani przetwarzanie transakcji, ani analityka. Bliżej mu do analityki, ponieważ proces wsadowy zwykle skanuje duże fragmenty wejściowego zbioru danych. Jednak przepływ pracy w modelu MapReduce różni się od zapytań SQL-owych używanych w analityce (zob. punkt „Porównanie Hadoopa do rozproszonych baz danych”). Dane wyjściowe z procesu wsadowego często nie są raportem i mają inną strukturę.

## **Tworzenie indeksów wyszukiwania**

Firma Google pierwotnie stosowała model MapReduce do budowania indeksów wyszukiwarki. Używano do tego przepływu pracy obejmującego 5 – 10 zadań MapReduce [1]. Choć Google potem przestał korzystać z modelu MapReduce w tym celu [43], można lepiej zrozumieć to narzędzie, przyglądając się mu z perspektywy budowania indeksu wyszukiwania. Nawet dziś MapReduce z Hadoopa pozostaje dobrym narzędziem do budowania indeksów dla systemów Lucene i Solr [44].

W punkcie „Wyszukiwanie pełnotekstowe i indeksy rozmyte” pokrótce opisano działanie indeksów pełnotekstowych, takich, jakie występują w systemie Lucene. Używany jest plik (słownik pojęć), w którym można wydajnie znaleźć określone słowo kluczowe i listę identyfikatorów wszystkich dokumentów zawierających to słowo (listę wystąpień). Jest to bardzo uproszczony obraz indeksu wyszukiwania. W praktyce potrzebne są różne dodatkowe dane, aby móc porządkować wyniki wyszukiwania według adekwatności, uwzględniać literówki i synonimy itd. Jednak zasada pozostaje taka sama.

Jeśli wykonujesz wyszukiwanie pełnotekstowe na stałym zbiorze dokumentów, proces wsadowy może być bardzo skutecznym sposobem budowania indeksów. Mapperzy dzielą zbiór dokumentów w odpowiedni sposób, każdy reducer buduje indeks dla swojej partycji, a pliki indeksu są zapisywane w rozproszonym systemie plików. Budowanie tego rodzaju indeksów dzielonych na podstawie dokumentów (zob. punkt „Podział na partycje a indeksy pomocnicze”) bardzo dobrze się nadaje do wykonywania równoległego.

Ponieważ zapytania o słowo kluczowe kierowane do indeksu wyszukiwania to operacja tylko do odczytu, pliki indeksu po utworzeniu są niemodyfikowalne.

Jeśli indeksowany zbiór dokumentów się zmienia, jedną z możliwości jest okresowe ponowne uruchamianie całego przepływu pracy i po jego ukończeniu zastępowanie wszystkich wcześniejszych plików indeksu nowymi plikami. To podejście może być obliczeniowo kosztowne, jeśli zmiana dotyczy tylko niewielkiej liczby dokumentów. Zaletą jest to, że proces indeksowania bardzo łatwo jest zrozumieć: podawane są dokumenty, a uzyskiwane indeksy.

Inna możliwość to przyrostowe budowanie indeksów. W rozdziale 3. opisano, że jeśli chcesz dodać, usunąć lub zaktualizować dokumenty występujące w indeksie, Lucene zapisuje nowe pliki segmentu oraz asynchronicznie scala i kompresuje te pliki w tle. Więcej o tego rodzaju przetwarzaniu przyrostowym dowiesz się z rozdziału 11.

## **Bazy kluczy i wartości jako dane wyjściowe z procesu wsadowego**

Indeksy wyszukiwania to tylko jedno z możliwych danych wyjściowych przepływu pracy z przetwarzaniem wsadowym. Innym częstym zastosowaniem przetwarzania wsadowego jest budowanie systemów uczenia maszynowego takich jak klasyfikatory (np. filtry antyspamowe, wykrywacze

anomalii, systemy rozpoznawania obrazu) i systemy rekomendacji (np. ludzi, których możesz znać, produktów, które mogą Cię interesować, lub powiązanych wyszukiwań [29]).

Danymi wyjściowymi z takich zadań wsadowych jest często baza — np. taka, która na podstawie identyfikatora użytkownika zwraca proponowanych znajomych tej osoby lub na podstawie identyfikatora produktu generuje listę powiązanych towarów [45].

Zapytania do takich baz są kierowane z aplikacji sieciowej, która obsługuje żądania użytkownika i jest zwykle niezależna od infrastruktury Hadoopa. Jak więc dane wyjściowe z procesu wsadowego mogą trafić ponownie do bazy, aby aplikacja sieciowa mogła kierować do niej zapytania?

Najbardziej oczywistym wyborem może się wydawać zastosowanie bezpośrednio w mapperze lub reducerze biblioteki klienckiej dla wybranej bazy i zapisywanie przez prace wsadowe danych od razu na serwerze bazodanowym rekord po rekordzie. To rozwiązanie działa (przy założeniu, że reguły zapory umożliwiają bezpośredni dostęp środowiska Hadoopa do produkcyjnych baz danych), jednak z kilku powodów jest niewłaściwe:

- Wcześniej w kontekście połączeń wyjaśniono, że zgłaszanie żądania sieciowego na potrzeby każdego rekordu jest o kilka rzędów wielkości wolniejsze niż standardowa przepustowość pracy wsadowej. Nawet jeśli biblioteka kliencka obsługuje operacje wsadowe, wydajność tego rozwiązania będzie zapewne niska.
- Prace w modelu MapReduce często równolegle wykonują wiele operacji. Gdyby wszystkie mappery i reducery jednocześnie zapisywały dane w tej samej bazie wyjściowej z szybkością typową dla procesów wsadowych, szybko nastąpiłoby przeciążenie bazy i zapewne spadłaby wydajność obsługi przez nią zapytań. To z kolei może prowadzić do problemów operacyjnych w innych częściach systemu [35].
- MapReduce w zakresie danych wyjściowych z pracy zwykle zapewnia gwarancje typu „wszystko albo nic”. Jeśli praca kończy się powodzeniem, wynik dla każdej operacji jest zwracany tylko raz — nawet gdy niektóre prace kończą się porażką i trzeba je ponowić. Jeżeli cała praca zawiedzie, nie są zwracane żadne dane wyjściowe. Jednak wykonywanie w pracy zapisu w zewnętrznym systemie powoduje widoczne na zewnątrz efekty uboczne, których nie da się ukryć w ten sposób. Dlatego trzeba uwzględnić ujawnianie wyników częściowo ukończonych zadań w innych systemach, a także złożone aspekty prób wykonania i spekulatywnego wykonywania zadań w Hadoopie.

Znacznie lepszym rozwiązaniem jest generowanie zupełnie nowej bazy *w ramach* pracy wsadowej i zapisywanie tej bazy jako plików w katalogu wyjściowym pracy w rozproszonym systemie plików (tak jak w przypadku indeksów wyszukiwania z poprzedniego punktu). Te pliki danych po zapisaniu stają się niemodyfikowalne i można je wczytywać masowo na serwery obsługujące zapytania z samym odczytem. Istnieją różne bazy z kluczami i wartościami obsługujące budowanie plików w pracach w modelu MapReduce. Te bazy to np.: Voldemort [46], Terrapin [47], ElephantDB [48] i HBase (metoda wczytywania masowego) [49].

Budowanie takich plików bazy danych jest dobrym zastosowaniem modelu MapReduce. Użycie mappera do pobrania kluczy i posortowania danych na ich podstawie to już duża część pracy potrzebnej do zbudowania indeksu. Ponieważ większość baz z kluczami i wartościami jest przeznaczona tylko do odczytu (plik może zostać zapisany tylko raz przez prace wsadowe, po czym staje

się niemodyfikowalny), używane struktury danych są stosunkowo proste. Nie jest potrzebny np. dziennik zapisu z wyprzedzeniem (zob. punkt „Zapewnianie niezawodności b-drzew”).

W trakcie wczytywania danych do bazy Voldemort serwer wciąż obsługuje żądania kierowane do dawnych plików z danymi. Nowe pliki z danymi są wtedy kopiowane z rozproszonego systemu plików na dysk lokalny serwera. Po zakończeniu kopiowania serwer atomowo przełącza się na obsługę zapytań za pomocą nowych plików. Jeśli wystąpią problemy, można łatwo ponownie się przełączyć na korzystanie z dawnych plików, ponieważ wciąż są one dostępne i niemodyfikowalne [46].

### Filozofia obsługi danych wyjściowych z procesów wsadowych

Omówiona wcześniej filozofia Uniksa (zob. punkt „Filozofia Uniksa”) zachęca do eksperymentów, ponieważ przepływ danych jest w niej bezpośrednio opisany. Program wczytuje dane wejściowe i zapisuje dane wyjściowe. W tym procesie dane wejściowe nie są modyfikowane, a wcześniejsze dane wyjściowe są w całości zastępowane nowymi. Efekty uboczne nie występują. To oznacza, że możesz dowolnie często ponawiać polecenia, dopracowywać je lub debugować bez naruszania stanu systemu.

Obsługa danych wyjściowych z zadań MapReduce jest oparta na tej samej filozofii. Dzięki traktowaniu danych wejściowych jako niemodyfikowalnych i unikaniu efektów ubocznych (takich jak zapis w zewnętrznych bazach) prace wsadowe nie tylko uzyskują wysoką wydajność, ale też są łatwiejsze w zarządzaniu:

- Jeśli spowodujesz błąd w kodzie i dane wyjściowe będą nieprawidłowe lub uszkodzone, możesz się cofnąć do poprzedniej wersji kodu i ponownie uruchomić pracę, a dane wyjściowe znów staną się poprawne. Jeszcze prostsze jest zachowanie dawnych danych wyjściowych w innym katalogu i przełączenie się na korzystanie z nich. Bazy z transakcjami z odczytem i zapisem nie mają tej cechy. Jeśli zastosujesz błędny kod, który zapisuje w bazie nieprawidłowe dane, wycofanie kodu nie naprawi danych z bazy. Możliwość przywrócenia stanu po użyciu nieprawidłowego kodu to *odporność na błędy ludzkie* [50].
- Dzięki łatwości wycofywania zmian można rozwijać funkcje znacznie szybciej niż w środowisku, gdzie błędy mogą powodować nieodwracalne szkody. Zasada minimalizowania nieodwracalności jest korzystna w zwinnym rozwoju oprogramowania [51].
- Jeśli zadanie mapowania lub redukcji zakończy się niepowodzeniem, MapReduce automatycznie ponownie je zaplanuje i uruchomi dla tych samych danych wejściowych. Jeżeli problem wynika z błędu w kodzie, będzie się powtarzać i ostatecznie doprowadzi do porażki po kilku próbach. Jednak gdy niepowodzenie to skutek przejściowego problemu, odporność na błędy zostaje zachowana. Automatyczne ponawianie prób jest bezpieczne tylko dzięki temu, że dane wejściowe są niemodyfikowalne, a dane wyjściowe z zakończonej niepowodzeniem pracy są przez MapReduce odrzucane.
- Ten sam zestaw plików można wykorzystać jako dane wejściowe w różnych pracach, w tym w pracach monitorujących, które obliczają wskaźniki i oceniają, czy dane wyjściowe mają oczekiwane cechy (np. przez porównanie ich z danymi wyjściowymi z poprzedniej pracy i zmierzenie rozbieżności).

- Prace w modelu MapReduce, podobnie jak narzędzia uniksowe, pozwalają oddzielić logikę od infrastruktury (konfiguracji katalogów wejściowych i wyjściowych). Skutkuje to podziałem zadań i pozwala na ponowne wykorzystanie kodu. Jeden zespół może się skoncentrować na implementowaniu pracy, która dobrze robi jedną rzecz, natomiast inne grupy mogą decydować o tym, gdzie i kiedy uruchamiać tę pracę.

W opisanych obszarach zasady projektowe sprawdzające się w Uniksie wydają się działać dobrze także w Hadoopie. Jednak Unix i Hadoop różnią się między sobą na kilka sposobów. Na przykład większość narzędzi uniksowych zakłada, że pobierane są nietylizowane pliki tekstowe. Dlatego konieczne jest rozbudowane parsowanie danych wejściowych (w przykładowej analizie dziennika na początku rozdziału do pobrania adresu URL posłużyło wyrażenie `{print $7}`). W Hadoopie niektóre z takich niskopoziomowych konwersji składniowych są eliminowane dzięki bardziej ustrukturyzowanym formatom plików. Często używane są Avro (zob. punkt „Avro”) i Parquet (zob. punkt „Bazy kolumnowe”), ponieważ zapewniają wydajne kodowanie oparte na schemacie i umożliwiają modyfikowanie schematów (zob. rozdział 4.).

## Porównanie Hadoopa z rozproszonymi bazami danych

Zobaczyłeś, że Hadoop przypomina rozproszoną wersję Uniksa, w której HDFS to system plików, a MapReduce to nietypowa implementacja procesu Uniksa (która między etapami mapowania i redukcji zawsze uruchamia narzędzie `sort`). Zobaczyłeś, jak za pomocą podstawowych elementów zaimplementować różne złączenia i operacje grupowania.

Gdy opublikowana została praca na temat modelu MapReduce [1], w pewnym sensie nie była to całkowita nowość. Wszystkie opisane w kilku ostatnich punktach algorytmy przetwarzania i równoległego złączania zostały już zaimplementowane w tzw. bazach **MPP** (ang. *massively parallel processing*) ponad dekadę wcześniej [3, 40]. Pierwsze rozwiązania z tego obszaru to maszyna bazodanowa Gamma, Teradata i Tandem NonStop SQL [52].

Największą różnicą jest to, że w bazach MPP liczy się głównie równoległe wykonywanie analitycznych zapytań SQL-owych w klastrze maszyn, natomiast połączenie modelu MapReduce z rozproszonym systemem plików [19] zapewnia coś, co bardziej przypomina system operacyjny ogólnego przeznaczenia, w którym można wykonywać dowolne programy.

### Różnorodność sposobu przechowywania

Bazy wymagają strukturyzowania danych zgodnie z określonym modelem (np. relacyjnym lub dokumentowym), natomiast pliki w rozproszonym systemie plików to tylko sekwencje bajtów, które można zapisać za pomocą dowolnego modelu danych i kodowania. Mogą to być kolekcje rekordów z bazy, ale równie dobrze mogą mieć postać tekstu, grafiki, filmów, odczytów z czujnika, macierzy rzadkich, wektorów cech, sekwencji genomu itd.

Ujmijmy to bezpośrednio: Hadoop umożliwił niewybiórcze umieszczanie danych w systemie HDFS i dopiero późniejsze ustalanie, jak je dalej przetwarzać [53]. Natomiast bazy MPP zwykle wymagają starannego wstępnego modelowania danych i wzorców zapytań przed zaimportowaniem danych w zastrzeżonym formacie używanym w określonej bazie.

Z perspektywy purysty może się wydawać, że to staranne modelowanie i importowanie jest korzystne, ponieważ oznacza, że użytkownicy bazy mogą pracować z danymi wyższej jakości. Jednak w praktyce szybkie udostępnianie danych (nawet zapisanych w niewygodnym, trudnym w użyciu nieprzetworzonym formacie) okazuje się często przydatniejsze niż próby wyboru z góry idealnego modelu danych [54].

Idea jest tu podobna jak w hurtowniach danych (zob. punkt „Hurtownie danych”). Już samo przeniesienie danych z różnych części dużej organizacji w jedno miejsce jest cenne, ponieważ umożliwia złączanie rozdzielonych wcześniej zbiorów danych. Staranne projektowanie schematu niezbędne w bazach MPP powoduje, że scentralizowany zapis danych jest wolniejszy. Zapis danych w nieprzetworzonej postaci i opracowywanie projektu schematu w późniejszym terminie pozwalają przyspieszyć zbieranie danych (ta metoda bywa czasem nazywana „jeziorem danych” — ang. *data lake* — lub centralą danych przedsiębiorstwa [55]).

Niewybiórczy zapis danych przenosi odpowiedzialność za ich interpretowanie. Zamiast zmuszać producenta zbioru danych do zapisu ich w standardowym formacie, interpretację danych traktuje się jako problem konsumenta (to podejście ustalania schematu na etapie odczytu; zob. punkt „Elastyczność schematu w modelu opartym na dokumentach”). Może to być korzystne, jeśli producentem i konsumentem są różne zespoły o innych priorytetach. Możliwe, że jeden idealny model danych nawet nie istnieje i że różne spojrzenia na dane są odpowiednie do innych celów. Proste zapisanie danych w nieprzetworzonej postaci pozwala na różne transformacje. To podejście zostało nazwane *zasadą sushi*: „surowe dane są lepsze” [57].

Dlatego Hadoop jest często używany do implementowania procesów ETL (zob. punkt „Hurtownie danych”). Dane z systemów przetwarzania transakcji są umieszczane w rozproszonym systemie plików w nieprzetworzonej postaci, a następnie należy napisać pracę w modelu MapReduce w celu uporządkowania danych, przekształcenia ich na postać relacyjną i zaimportowania do hurtowni danych MPP na potrzeby analityki. Także tu przeprowadzane jest modelowanie danych, jednak odbywa się to w odrębnym kroku, niezależnie od zbierania danych. Ten podział jest możliwy, ponieważ rozproszony system plików obsługuje dane zakodowane w dowolnym formacie.

### Różnorodność modeli przetwarzania

Bazy MPP to monolityczne, ściśle zintegrowane systemy, które odpowiadają za układ danych na dysku oraz planowanie, szeregowanie i wykonywanie zapytań. Ponieważ te aspekty można dostroić i zoptymalizować pod kątem konkretnych wymagań bazy, cały system może uzyskać bardzo wysoką wydajność dla typów zapytań, z myślą o których został zaprojektowany. Ponadto język zapytań SQL pozwala pisać zwięzłe zapytania i zapewnia elegancką semantykę bez konieczności pisania kodu. Dzięki temu może być używany w narzędziach graficznych (takich jak Tableau) stosowanych przez analityków biznesowych.

Jednak nie wszystkie rodzaje przetwarzania można zapisać w formie zapytań SQL-owych. Jeśli np. budujesz systemy uczenia maszynowego i rekomendacji, tworzysz indeksy pełnotekstowe z modelami porządkowania elementów na podstawie adekwatności lub przeprowadzasz analizę zdjęć, zapewne będziesz potrzebował bardziej ogólnego modelu przetwarzania danych. Przetwarzanie tego rodzaju jest często specyficzne dla konkretnej aplikacji (np. inżynieria cech w uczeniu maszynowym, modele języka naturalnego w tłumaczeniach maszynowych lub funkcje szacowania ryzyka w wykrywaniu oszustw), dlatego nieuniknione jest pisanie kodu zamiast samych zapytań.

MapReduce dał inżynierom możliwość łatwego uruchamiania własnego kodu na dużych zbiorach danych. Za pomocą systemu HDFS i modelu MapReduce *możesz* zbudować system przetwarzania zapytań SQL-owych. To właśnie zrobiono w projekcie Hive [31]. Ponadto możesz też pisać procesy wsadowe innego rodzaju, których nie da się przedstawić w formie zapytań SQL-owych.

Ostatecznie użytkownicy stwierdzili, że MapReduce jest zbyt ograniczający i dla niektórych typów przetwarzania ma zbyt niską wydajność. Dlatego na podstawie Hadoopa opracowano różne inne modele przetwarzania (niektóre z nich poznasz w punkcie „Poza model MapReduce”). Dostępność dwóch modeli przetwarzania, SQL-a i MapReduce, nie wystarczyła. Potrzebne były jeszcze inne modele. A dzięki otwartości Hadoopa można było zaimplementować cały zestaw podejść, które byłyby niemożliwe w ramach monolitycznej bazy MPP [58].

Bardzo ważne jest to, że wszystkie te modele przetwarzania można stosować w jednym współużytkowanym klastrze maszyn i wszystkie mogą korzystać z tych samych plików z rozproszonego systemu plików. Gdy używasz Hadoopa, nie musisz importować danych do kilku odmiennych wyspecjalizowanych systemów na potrzeby poszczególnych metod przetwarzania. Ten system jest wystarczająco elastyczny, aby zapewnić obsługę różnego obciążenia roboczego w tym samym klastrze. Ponieważ nie trzeba przenosić danych, można znacznie łatwiej generować wartości na ich podstawie. Dużo łatwiejsze jest też eksperymentowanie z nowymi modelami przetwarzania.

Ekosystem Hadoopa obejmuje zarówno bazy OLTP z dostępem losowym, takie jak HBase (zob. punkt „Pliki SSTable i drzewa LSM”), jak i bazy analityczne w stylu MPP, takie jak Impala [41]. Ani HBase, ani Impala nie używają modelu MapReduce, ale obie przechowują dane w systemie HDFS. W obu stosowane są zupełnie odmienne podejścia do przetwarzania i używania danych, jednak bazy te mogą współistnieć w jednym systemie i można je w nim zintegrować.

## Projektowanie z uwzględnieniem częstych błędów

W trakcie porównywania modelu MapReduce z bazami MPP widoczne są też dwie inne różnice w projekcie związane z obsługą błędów oraz wykorzystaniem pamięci i dysku. Procesy wsadowe są mniej wrażliwe na błędy niż systemy online, ponieważ niepowodzenie procesu wsadowego nie wpływa natychmiast na użytkowników i zawsze można ponownie uruchomić taki proces.

Awaria węzła w trakcie przetwarzania zapytania powoduje w większości baz MPP anulowanie całego zapytania, po czym albo użytkownik może ponownie zgłosić zapytanie, albo jest ono ponawiane automatycznie [3]. Zapytania zwykle działają kilka sekund (lub najwyżej kilka minut), dlatego ten sposób obsługi błędów jest akceptowalny, ponieważ koszty ponawiania prób nie są wysokie. W bazach MPP preferuje się utrzymywanie jak największej ilości danych w pamięci (np. za pomocą złączania z haszowaniem), aby uniknąć kosztów odczytu danych z dysku.

Z kolei MapReduce może sobie radzić z awariami zadań mapowania lub redukcji bez wpływu na całą pracę, ponawiając pracę na poziomie poszczególnych zadań. MapReduce bardzo często zapisuje dane na dysku — po części w celu zapewnienia odporności na błędy, a po części z powodu założenia, że zbiór danych będzie za duży, aby zmieścić go w pamięci.

Podejście stosowane w modelu MapReduce lepiej nadaje się dla większych prac, które przetwarzają tak dużo danych i działają tak długo, że zapewne doświadczą niepowodzenia przynajmniej jednego zadania. Wtedy ponowne uruchomienie całej pracy z powodu niepowodzenia jednego zadania by-



łoby marnotrawstwem. Nawet jeśli przywracanie stanu na poziomie pojedynczego zadania powoduje koszty sprawiające, że przetwarzanie wolne od błędów staje się wolniejsze, może to być uzasadnionym kompromisem, jeżeli niepowodzenie zadań zdarza się wystarczająco często.

Jak bardzo realistyczne są jednak te założenia? W większości klastrów zdarzają się awarie maszyn, choć jest to dość rzadkie. Zapewne na tyle rzadkie, że większość prac się z tym nie zetknie. Czy warto ponosić znaczne koszty w celu uzyskania odporności na błędy?

Aby zrozumieć oszczędne korzystanie z pamięci i przywracanie stanu na poziomie zadań w MapReduce, warto się przyjrzeć środowisku, na potrzeby którego pierwotnie zaprojektowano ten model. Google korzysta z centrów danych o zróżnicowanym zastosowaniu, w których usługi produkcyjne w trybie online i prace wsadowe w trybie offline działają na tych samych maszynach. Każde zadanie ma przydzielone zasoby (rdzenie procesora, pamięć RAM, miejsce na dysku itd.) określane za pomocą kontenerów. Każde zadanie ma też ustalony priorytet. Gdy zadanie o wyższym priorytecie wymaga więcej zasobów, zadania o niższym priorytecie z tej samej maszyny mogą zostać zakończone (wywłaszczone) w celu zwolnienia zasobów. Priorytety wpływają też na cenę zasobów obliczeniowych. Zespoły muszą płacić za wykorzystywane zasoby, a procesy o wyższych priorytetach kosztują więcej [59].

Ta architektura pozwala na przyznawanie nadmiarowej ilości zasobów obliczeniowych dla zadań nieprodukcyjnych (o niskim priorytecie), ponieważ system wie, że w razie potrzeby może odzyskać te zasoby. Przydział nadmiarowej ilości zasobów pozwala na lepsze wykorzystanie możliwości maszyn i wyższą wydajność w porównaniu z systemami, w których zadania produkcyjne są oddzielone od nieprodukcyjnych. Jednak ponieważ prace w modelu MapReduce działają z niskim prioryteciem, są stale zagrożone wywłaszczeniem, ponieważ proces o wyższym priorytecie może wymagać ich zasobów. Zadania wsadowe „dostają resztki ze stołu” i wykorzystują zasoby obliczeniowe pozostałe po tym, jak procesy o wysokim priorytecie wykorzystały to, co jest im potrzebne.

W Google’u dla zadania MapReduce działającego przez godzinę ryzyko zamknięcia w celu zrobienia miejsca procesowi o wyższym priorytecie wynosi ok. 5%. Ten poziom jest o ponad rząd wielkości wyższy niż częstotliwość niepowodzeń spowodowanych błędem sprzętowym, restartem maszyny lub innymi przyczynami [59]. Przy wywłaszczaniu na tym poziomie w pracy o 100 zadaniach, z których każde działa przez 10 minut, prawdopodobieństwo zamknięcia przynajmniej jednego z nich przed zakończeniem wynosi ponad 50%.

To dlatego MapReduce został tak zaprojektowany, by był odporny na częste nieoczekiwane zakończenie zadania. Taki projekt wynika nie z wysokiej zawodności sprzętu, a z tego, że swoboda zamykania procesów pozwala lepiej wykorzystać zasoby w klastrze obliczeniowym.

W otwartych programach szeregujących dla klastrów wywłaszczanie jest stosowane rzadziej. CapacityScheduler w usłudze YARN obsługuje wywłaszczanie na potrzeby równoważenia alokacji zasobów w różnych kolejkach [58], jednak w czasie, gdy powstaje ta książka, w narzędziach YARN, Mesos i Kubernetes wywłaszczanie na podstawie priorytetów nie jest możliwe [60]. W środowisku, którym zadania są zamykane rzadziej, decyzje projektowe podjęte w MapReduce mają mniej sensu. W następnym punkcie poznasz rozwiązania inne niż MapReduce, oparte na odmiennych decyzjach projektowych.

# Poza model MapReduce

Choć pod koniec pierwszej dekady XXI w. MapReduce stał się bardzo popularny i zyskał duży rozgłos, jest to tylko jeden z wielu możliwych modeli programowania dla systemów rozproszonych. W zależności od ilości danych, ich struktury i sposobu przetwarzania inne narzędzia mogą się okazać bardziej odpowiednie do zapisu obliczeń.

W tym rozdziale wiele miejsca poświęcono modelowi MapReduce, ponieważ jest on przydatny edukacyjnie — jest stosunkowo przejrzystą i prostą abstrakcją opartą na rozproszonym systemie plików. *Prostą* w tym sensie, że można zrozumieć, jak działa. Nie chodzi o to, że MapReduce jest łatwy w użyciu — wprost przeciwnie. Implementowanie złożonej pracy za pomocą samych interfejsów API modelu MapReduce jest dość trudne i pracochłonne. Musisz np. implementować wszystkie algorytmy złączania od podstaw [37].

W reakcji na trudności z bezpośrednim używaniem modelu MapReduce opracowano różne wysokopoziomowe modele programowania (Pig, Hive, Cascading, Crunch). Są to abstrakcje oparte na MapReduce. Jeśli rozumiesz, jak działa MapReduce, dość łatwo opanujesz te modele, a ich wysokopoziomowe konstrukty znacznie upraszczają implementowanie wielu standardowych zadań z obszaru przetwarzania wsadowego.

Jednak problemy dotyczą też samego modelu wykonywania modelu MapReduce, których nie da się wyeliminować, dodając warstwę abstrakcji. Przejawiają się one w formie niskiej wydajności niektórych rodzajów przetwarzania. Z jednej strony MapReduce jest bardzo solidny. Można zastosować go do przetwarzania niemal dowolnie dużych ilości danych w zawodnym systemie z wieloma użytkownikami i częstym zamykaniem zadań, a i tak wykona swoją pracę (choć powoli). Z drugiej strony dla niektórych rodzajów przetwarzania istnieją narzędzia działające o rzędy wielkości szybciej.

W pozostałej części rozdziału przedstawiono wybrane z tych innych metod przetwarzania wsadowego. W rozdziale 11. omówiono przetwarzanie strumieniowe, które można uznać za następny sposób na przyspieszenie przetwarzania wsadowego.

## Materializowanie stanu pośredniego

Wcześniej opisano, że każda praca w modelu MapReduce jest niezależna od wszystkich pozostałych. Głównym punktem styczności pracy z resztą świata są katalogi wejściowy i wyjściowy w rozproszonym systemie plików. Jeśli chcesz, by dane wyjściowe z jednej pracy były danymi wejściowymi drugiej, musisz w drugiej pracy ustawić katalog wejściowy na katalog wyjściowy pierwszej. Ponadto zewnętrzny program szeregujący dla przepływu pracy musi uruchamiać drugą pracę dopiero po zakończeniu pierwszej.

Ta konfiguracja jest sensowna, jeśli dane wyjściowe z pierwszej pracy to zbiór danych, który chcesz udostępnić w organizacji. Wtedy potrzebujesz możliwości wskazywania tego zbioru na podstawie nazwy i ponownego wykorzystania go jako wejścia w kilku innych pracach (w tym w pracach utworzonych przez inne zespoły). Publikowanie danych w znanej lokalizacji w rozproszonym systemie plików pozwala na luźne powiązanie komponentów, dzięki czemu prace nie muszą wiedzieć, kto generuje ich dane wejściowe lub kto korzysta z ich danych wyjściowych (zob. punkt „Oddzielanie logiki od infrastruktury”).

Jednak w wielu sytuacjach wiadomo, że dane wyjściowe jednej pracy są używane wyłącznie jako dane wejściowe innej, zarządzanej przez ten sam zespół. Wtedy pliki w rozproszonym systemie plików są tylko *stanem pośrednim*. Służą do przekazywania danych z jednej pracy do następnej. W złożonym przepływie pracy służącym do budowania systemów rekomendacji i obejmującym 50 lub 100 prac w modelu MapReduce [29] taki stan pośredni pojawia się często.

Proces zapisu stanu pośredniego w plikach to *materializowanie*. To pojęcie pojawiło się wcześniej w kontekście widoków materializowanych w punkcie „Agregowanie — kostki danych i widoki zmaterializowane”. Oznacza ono wcześnie obliczanie wyniku jakiejś operacji i zapisywanie go zamiast obliczania go na żądanie.

Z kolei w przykładowej analizie dziennika z początku tego rozdziału zastosowano potoki Uniksa do połączenia danych wyjściowych z jednego polecenia z danymi wejściowymi z następnego. Potoki nie materializują w pełni stanu pośredniego, a zamiast tego stopniowo *strumieniują* wyjście do wejścia, używając tylko małego bufora w pamięci.

Pełne materializowanie stanu pośredniego w modelu MapReduce ma pewne wady w porównaniu z potokami Uniksa:

- Praca w modelu MapReduce może rozpocząć działanie dopiero po ukończeniu wszystkich zadań z wcześniejszych prac (generujących jej dane wejściowe), natomiast procesy połączone potokiem Uniksa są uruchamiane w tym samym czasie, a dane wyjściowe są wykorzystywane bezpośrednio po ich wygenerowaniu. Asymetria lub nierównomierne obciążenie w różnych maszynach sprawiają, że w pracy często występuje kilka opóźnionych zadań, których przetwarzanie trwa znacznie dłużej niż pozostałych. Konieczność oczekiwania na ukończenie wszystkich wcześniejszych zadań spowalnia przebieg całego przepływu pracy.
- Mapperzy są często nadmiarowe. Wczytują ten sam plik, który został właśnie zapisany przez reducer, i przygotowują go do następnego etapu podziału i sortowania. W wielu sytuacjach mapper mógłby być częścią poprzedniego reducera. Gdyby wyjście reducera było dzielone i sortowane w taki sam sposób jak wyjście mappera, reducery można byłoby bezpośrednio połączyć w łańcuch, bez przeplatania ich mapperami.
- Zapisywanie stanu pośredniego w rozproszonym systemie plików oznacza, że pliki są replikowane w kilku węzłach, co jest często niepotrzebne dla tego typu danych tymczasowych.

## Systemy przepływu danych

Aby rozwiązać opisane problemy z modelem MapReduce, opracowano kilka nowych systemów wykonawczych dla rozproszonych obliczeń wsadowych. Najbardziej znane z tych systemów to: Spark [61, 62], Tez [63, 64] i Flink [65, 66]. Różnią się one projektami, jednak mają jedną cechę wspólną: obsługują cały przepływ pracy jako jedną pracę, zamiast dzielić go na niezależne prace podrzędne.

Ponieważ te systemy bezpośrednio modelują przepływ danych przez kilka etapów przetwarzania, są nazywane *systemami przepływu danych*. Podobnie jak MapReduce wielokrotnie wywołują zdefiniowaną przez użytkownika funkcję, aby przetwarzać rekordy jeden po drugim w jednym wątku. Równoległość wynika z podziału danych wejściowych, a systemy kopiuje dane wyjściowe z jednej funkcji przez sieć, aby wykorzystać je jako dane wejściowe innej funkcji.

Inaczej niż w MapReduce funkcje nie odgrywają ściśle ról mapperów i reducerów. Zamiast tego można je łączyć w bardziej swobodny sposób. Te funkcje są nazywane *operatorami*, a system przepływu danych udostępnia kilka możliwości łączenia danych wyjściowych jednego operatora z danymi wejściowymi innego:

- Jedną z możliwości to ponowny podział i sortowanie rekordów według klucza, tak jak na etapie tasowania w MapReduce (zob. punkt „Rozproszone wykonywanie operacji w MapReduce”). Ten mechanizm umożliwia grupowanie oraz złączanie z sortowaniem przez scalanie działające tak jak w MapReduce.
- Inną możliwością to użycie kilku wejść i podział ich w standardowy sposób, ale z pominięciem sortowania. Pozwala to przyspieszyć złączenia z podziałem i haszowaniem, gdzie podział rekordów jest ważny, ale ich kolejność nie ma znaczenia, ponieważ w tablicy z haszowaniem uporządkowanie i tak jest losowe.
- W złączaniu z rozsyłaniem i haszowaniem te same dane wyjściowe z jednego operatora można przesyłać do wszystkich partycji operatora odpowiedzialnego za złączenie.

Ten rodzaj systemów przetwarzania jest oparty na systemach akademickich takich jak Dryad [67] i Nephelē [68]. Ma on kilka zalet w porównaniu z MapReduce:

- Kosztowne operacje takie jak sortowanie trzeba wykonywać tylko wtedy, gdy jest to konieczne. Nie są one domyślnie wykonywane zawsze między wszystkimi etapami mapowania i redukcji.
- Nie występują tu niepotrzebne zadania mapowania, ponieważ działania mappera często można umieścić we wcześniejszym operatorze redukcji (ponieważ mapper nie zmienia podziału zbioru danych).
- Ponieważ wszystkie złączenia i zależności między danymi są bezpośrednio zadeklarowane w przepływie pracy, program szeregujący wie, jakie dane są potrzebne w poszczególnych miejscach. Pozwala to na optymalizacje związane z lokalnością. Program szeregujący może np. próbować umieścić zadanie wykorzystujące pewne dane w tej samej maszynie, na której działa zadanie generujące te dane. Dane można wtedy przekazywać za pomocą współużytkowanego bufora pamięci zamiast kopiować je przez sieć.
- Zwykle wystarcza, by stan pośredni był przechowywany w pamięci lub zapisywany na dysku lokalnym. Wymaga to mniej operacji wejścia-wyjścia niż zapisywanie go w systemie HDFS (gdzie wymaga replikacji na kilka maszyn i zapisu na dysku w każdej replice). W MapReduce ta optymalizacja jest stosowana do danych wyjściowych mappera, jednak w systemach przepływu danych ta idea została uogólniona na wszystkie rodzaje stanu pośredniego.
- Operatory mogą rozpocząć działanie od razu, gdy ich dane wejściowe są gotowe. Nie trzeba czekać na ukończenie całego poprzedniego etapu przed rozpoczęciem następnego.
- Istniejące procesy maszyny JVM można ponownie wykorzystać do wykonywania nowych operatorów. W porównaniu z modelem MapReduce (który dla każdego zadania uruchamia nową maszynę JVM) zmniejsza to koszty rozruchu.

Systemy przepływu danych możesz wykorzystać do zaimplementowania tych samych obliczeń co w przepływach pracy w MapReduce. Dzięki opisanym tu optymalizacjom obliczenia zwykle są wtedy znacznie szybsze. Ponieważ operatory są uogólnieniem mapperów i reducerów, ten sam kod można uruchamiać w dowolnym systemie wykonawczym. Zaimplementowane w narzędziach Pig, Hive

lub Cascading przepływy pracy używające modelu MapReduce można za pomocą prostej zmiany konfiguracyjnej (bez modyfikowania kodu) przełączyć na system Tez lub Spark [64].

Tez to stosunkowo prosta biblioteka oparta na usłudze tasowania YARN. Służy do kopiowania danych między węzłami [58]. Z kolei Spark i Flink to rozbudowane platformy obejmujące własną warstwę komunikacji w sieci, program szeregujący i interfejsy API do komunikacji z użytkownikami. Te wysokopoziomowe interfejsy API opisano dalej.

## Odporność na błędy

Zaletą pełnej materializacji stanu pośredniego w systemie plików jest jego utrwalenie. Dzięki temu model MapReduce pozwala dość łatwo zapewnić odporność na błędy. Jeśli zadanie zakończy się niepowodzeniem, można je ponownie uruchomić na innej maszynie i ponownie wczytać te same dane wejściowe z systemu plików.

Spark, Flink i Tez unikają zapisu stanu pośredniego w systemie HDFS, dlatego zapewnianie odporności na błędy wygląda w nich inaczej. Jeśli nastąpi awaria maszyny i stan pośredni zostanie utracony, jest on ponownie obliczany na podstawie innych, wciąż dostępnych danych (jeżeli jest to możliwe, są to dane z wcześniejszego etapu pośredniego; w przeciwnym razie używa się pierwotnych danych wejściowych, zapisanych zwykle w systemie HDFS).

Aby umożliwić ponowne obliczenia, platforma musi śledzić, jak uzyskano określone dane — jakie partycje wejściowe i które operatory zostały użyte. Spark do rejestrowania pochodzenia danych używa abstrakcji **RDD** (ang. *resilient distributed dataset*), natomiast Flink zapisuje punkty kontrolne ze stanem operatora, co umożliwia wznowienie działania operatora po wystąpieniu błędu w jego wykonywaniu [66].

W trakcie ponownego generowania danych trzeba wiedzieć, czy ten proces jest *deterministyczny*, czyli czy na podstawie tych samych danych wejściowych operatory zawsze generują te same dane wyjściowe. Ta kwestia ma znaczenie, jeśli część utraconych danych została już przesłana do dalszych operatorów. Jeżeli operator jest ponownie uruchamiany i ponownie generowane dane nie są identyczne z pierwotnymi, utraconymi informacjami, dalszym operatorom bardzo trudno jest uwzględnić rozbieżności między dawnymi i nowymi danymi. W przypadku operatorów niedeterministycznych rozwiązanie polega zwykle na zamknięciu także dalszych operatorów i ponownym uruchomieniu ich dla nowych danych.

Aby uniknąć tego rodzaju kaskadowych błędów, lepiej jest pisać operatory deterministyczne. Zauważ jednak, że łatwo mogą się przypadkowo pojawić niedeterministyczne operacje. Na przykład w wielu językach programowania nie ma gwarancji zachowania kolejności w trakcie iteracyjnego pobierania elementów tablicy z haszowaniem, wiele probabilistycznych i statystycznych algorytmów bezpośrednio wykorzystuje liczby losowe, a każde użycie zegara systemowego lub zewnętrznych źródeł danych jest niedeterministyczne. Takie przyczyny braku determinizmu trzeba wyeliminować, aby móc niezawodnie przywracać stan po awariach. Możesz np. generować liczby pseudolosowe za pomocą stałego ziarna.

Przywracanie stanu po awariach za pomocą ponownego obliczania danych nie zawsze jest właściwe. Jeśli dane pośrednie są znacznie mniejsze niż źródłowe lub gdy przetwarzanie jest bardzo wymagające obliczeniowo, prawdopodobnie taniej będzie materializować dane pośrednie w plikach, niż ponownie je generować.

## Dyskusja na temat materializowania

Wróćmy do analogii z Uniksem. Zobaczyłeś, że MapReduce to odpowiednik zapisu danych wyjściowych z każdego polecenia w pliku tymczasowym, natomiast systemy przepływu danych są dużo bardziej podobne do potoków Uniksa. Zwłaszcza Flink jest oparty na wykonywaniu potokowym — na stopniowym przekazywaniu danych wyjściowych z operatora do innych operatorów bez oczekiwania na uzyskanie gotowych danych wejściowych przed rozpoczęciem ich przetwarzania.

Sortowanie zawsze wymaga wykorzystania wszystkich danych wejściowych przed zwróceniem danych wyjściowych, ponieważ możliwe, że klucz z ostatniego rekordu wejściowego będzie najniższy, dlatego będzie musiał być pierwszym rekordem wyjściowym. Każdy operator wymagający sortowania musi więc zapisywać stan (przynajmniej przejściowo). Jednak wiele innych fragmentów przepływu pracy można wykonywać potokowo.

Gdy praca zostaje ukończona, jej dane wyjściowe trzeba zapisać w trwałym miejscu, tak by użytkownicy mogli je znaleźć i korzystać z nich. Zwykle dane zapisuje się ponownie w rozproszonym systemie plików. Dlatego gdy używasz systemu przepływu danych, wejściem oraz ostatecznym wyjściem pracy i tak są zwykle zmaterializowane zbiory danych w systemie HDFS. Podobnie jak w MapReduce dane wejściowe są niemodyfikowalne, a dane wyjściowe są zastępowane w całości. Korzyścią w porównaniu z MapReduce jest to, że nie trzeba zapisywać w systemie plików także wszystkich stanów pośrednich.

## Grafy i przetwarzanie iteracyjne

W punkcie „Modele danych przypominające graf” omówiono modelowanie danych z użyciem grafów oraz stosowanie języków zapytań dla grafów do poruszania się między krawędziami i wierzchołkami grafu. Informacje z rozdziału 2. dotyczyły głównie modelu OLTP — szybko wykonywanych zapytań wyszukujących niewielką liczbę wierzchołków zgodnych z określonymi kryteriami.

Warto też przyjrzeć się grafom w kontekście przetwarzania wsadowego, gdzie celem są przetwarzanie w trybie offline lub analizy całego grafu. Jest to często potrzebne w aplikacjach z obszaru uczenia maszynowego, np. w systemach rekomendacji lub klasyfikacji. Na przykład jednym z najbardziej znanych algorytmów do analizy grafów jest PageRank [69], który próbuje oszacować popularność strony WWW na podstawie tego, na jakich innych stronach znajdują się odsyłacze do niej. Tego algorytmu używa się we wzorze określającym kolejność wyświetlania wyników przez wyszukiwarkę.



Systemy przepływu danych takie jak Spark, Flink i Tez (zob. punkt „Materializowanie stanu pośredniego”) zwykle porządkują operatory w pracy za pomocą skierowanego grafu acyklicznego. To nie to samo co przetwarzanie grafów. W systemach przepływu danych to *przepływ danych między operatorami* jest oparty na grafie, natomiast same dane to zwykle typowe dla modelu relacyjnego krotki. Przy przetwarzaniu grafów to *same dane* mają postać grafu. Jest to następna myląca nazwa.

Wiele algorytmów dla grafów działa, poruszając się po pojedynczych krawędziach łączących przyległe wierzchołki i przekazując jakieś informacje. Proces ten może być powtarzany do momentu spełnienia jakiegoś warunku — np. odwiedzenia wszystkich krawędzi lub uzyskania określonej wartości wskaźnika. Przykład pokazano na rysunku 2.6, gdzie lista wszystkich lokalizacji w Ameryce

Północnej dostępnych w bazie jest generowana w wyniku wielokrotnego wybierania krawędzi określających lokalizacje wewnątrz innych lokalizacji (tak działa algorytm *domknięcia przechodniego*).

Można zapisać graf w rozproszonym systemie plików (w plikach z listami wierzchołków i krawędzi), jednak modelu „powtarzaj do momentu zakończenia” nie da się zapisać w samym modelu MapReduce, ponieważ przetwarza on dane tylko raz. Dlatego tego typu algorytmy są często implementowane *iteracyjnie*:

1. Zewnętrzny program szeregujący uruchamia proces wsadowy wykonujący jeden krok algorytmu.
2. Gdy proces wsadowy kończy działanie, program szeregujący sprawdza, czy to już koniec całej operacji. Odbywa się to na podstawie warunku zakończenia — np. nie ma już więcej krawędzi lub zmiana w porównaniu z poprzednią iteracją jest mniejsza od ustalonego progu.
3. Jeśli cała operacja nie została zakończona, program szeregujący wraca do kroku 1. i ponownie uruchamia proces wsadowy.

To podejście działa, ale jego implementacja w MapReduce jest często wysoce niewydajna, ponieważ MapReduce nie uwzględnia iteracyjnej natury tego algorytmu. MapReduce zawsze wczytuje cały wejściowy zbiór danych i generuje zupełnie nowy wyjściowy zbiór danych — nawet jeśli tylko niewielka część grafu zmieniła się w porównaniu z poprzednią iteracją.

### Model przetwarzania z systemu Pregel

Popularność zyskał model **BSP** (ang. *bulk synchronous parallel*) [70] stosowany jako optymalizacja wsadowego przetwarzania grafów. Został on zaimplementowany m.in. w narzędziach Apache Giraph [37], GraphX API w systemie Spark i Gelly API w systemie Flink [71]. Inna nazwa to model z systemu *Pregel*, ponieważ to podejście w kontekście przetwarzania grafów zostało spopularyzowane dzięki artykułowi na temat systemu Pregel firmy Google [72].

Przypomnij sobie, że w MapReduce mappery „przesyłają komunikat” do określonego wywołania reducera, ponieważ ten model łączy wszystkie wyjścia mappera o tym samym kluczu. Na podobnej zasadzie działa Pregel — jeden wierzchołek może „przesłać komunikat” do innego. Te komunikaty zwykle są przysyłane krawędziami grafu.

W każdej iteracji dla wszystkich wierzchołków wywoływana jest funkcja przekazująca wszelkie przesłane do danego wierzchołka komunikaty. Przypomina to wywołanie reducera. Różnica w porównaniu z MapReduce polega na tym, że w Preglu wierzchołek między iteracjami zachowuje stan w pamięci. Dlatego funkcja musi przetwarzać tylko nowe odebrane komunikaty. Jeśli do danej części grafu nie przesłano żadnych komunikatów, nie trzeba wykonywać tam żadnych operacji.

Jeśli potraktować każdy wierzchołek jak aktora, przypomina to nieco model oparty na aktorach (zob. punkt „Platformy z rozproszonymi aktorami”). Różnica polega na tym, że stan wierzchołka i komunikaty przekazywane między wierzchołkami są odporne na błędy i trwałe, a komunikacja odbywa się w ustalonych etapach: w każdej iteracji platforma dostarcza wszystkie komunikaty przesłane w poprzedniej iteracji. Aktory zwykle nie oferują takich gwarancji czasowych.

## Odporność na błędy

To, że wierzchołki mogą się komunikować wyłącznie poprzez przekazywanie komunikatów (a nie przez bezpośrednie kierowanie do siebie zapytań), pomaga poprawić wydajność prac w Preglu, ponieważ komunikaty można łączyć w porcje, a oczekiwanie na komunikację jest krótsze. Oczekiwanie następuje tylko między iteracjami. Ponieważ model z Pregla gwarantuje, że wszystkie komunikaty przesłane w jednej iteracji zostaną dostarczone w następnej, poprzednia iteracja musi w pełni zakończyć działania, a wszystkie wysłane w niej komunikaty muszą zostać skopiowane przez sieć; dopiero wtedy może się rozpocząć następna iteracja.

Choć komunikaty w sieci mogą zostać zgubione, zduplikowane lub dowolnie opóźnione (zob. punkt „Zawodne sieci”), implementacje modelu z Pregla gwarantują, że komunikaty zostaną przetworzone tylko raz w docelowym wierzchołku w następnej iteracji. Ta platforma, podobnie jak MapReduce, automatycznie przywraca stan po błędach, aby uprościć model programowania algorytmów opartych na Preglu.

Odporność na błędy jest uzyskiwana dzięki okresowemu zapisywaniu pod koniec iteracji punktów kontrolnych ze stanem wszystkich wierzchołków (czyli zapisywaniu całego stanu w trwałej pamięci). Jeśli węzeł zawiedzie i przechowywany w pamięci stan zostanie utracony, najprostszym rozwiązaniem będzie wycofać wszystkie operacje na grafie do ostatniego punktu kontrolnego i wznowić obliczenia. Jeżeli algorytm działa deterministycznie i komunikaty są rejestrowane, można też wybiórczo odzyskać tylko utraconą partycję (co opisano wcześniej w kontekście systemów przepływu danych) [72].

## Wykonywanie równoległe

Wierzchołek nie musi wiedzieć, w której fizycznej maszynie działa. Gdy przesyła komunikaty do innych wierzchołków, przekazuje je do identyfikatora wierzchołka. To platforma odpowiada za podział grafu (czyli decydowanie, które wierzchołki mają działać w poszczególnych maszynach) i przekierowywanie komunikatów w sieci, tak by dotarły w odpowiednie miejsce.

Ponieważ ten model programowania uwzględnia tylko jeden wierzchołek naraz (jest to nazywane „myśleniem z perspektywy wierzchołka”), platforma może podzielić graf w dowolny sposób. Idealny podział jest taki, że wierzchołki znajdują się w tej samej maszynie, jeśli muszą się często komunikować. Jednak znalezienie takiego zoptymalizowanego podziału jest trudne. W praktyce graf jest zwykle dzielony na podstawie arbitralnie przypisanych identyfikatorów bez prób grupowania powiązanych wierzchołków.

Wskutek tego algorytmy dla grafów często prowadzą do wysokich kosztów komunikacji między maszynami, a stan pośredni (komunikaty przesyłane między węzłami) zajmuje nieraz więcej miejsca niż pierwotny graf. Koszty przesyłania komunikatów w sieci mogą znacznie obniżyć wydajność rozproszonych algorytmów dla grafów.

Z tego powodu jeśli graf mieści się w pamięci jednego komputera, algorytm dla pojedynczej maszyny (możliwe, że nawet jednowątkowej) będzie wydajniejszy od rozproszonych procesów wsadowych [73, 74]. Nawet gdy wielkość grafu przekracza pojemność pamięci, ale może się on zmieścić na dyskach jednego komputera, możliwym rozwiązaniem jest przetwarzanie na jednej maszynie z użyciem platformy takiej jak GraphChi [75]. Jeśli graf jest za duży, aby zmieścić go na jednej



maszynie, nieuniknione jest stosowanie rozwiązania rozproszonego takiego jak model z Pregla. Wydajne równoległe wykonywanie algorytmów dla grafów jest obszarem, nad którym prowadzi się badania [76].

## Wysokopoziomowe interfejsy API i języki

W latach, jakie upłynęły od czasu spopularyzowania modelu MapReduce, systemy wykonawcze do rozproszonego przetwarzania wsadowego dojrzały. Obecnie infrastruktura jest na tyle stabilna, że umożliwia składowanie i przetwarzanie wielu petabajtów danych w klastrach obejmujących ponad 10 tys. maszyn. Ponieważ problem fizycznego wykonywania procesów wsadowych na taką skalę uznano za w mniejszym lub większym stopniu rozwiązany, uwagę skierowano na inne obszary: ulepszanie modelu programowania, poprawę wydajności przetwarzania i poszerzenie zbioru problemów, jakie można rozwiązać za pomocą omawianych technologii.

Wcześniej wyjaśniono, że języki i interfejsy API wyższego poziomu, np. Hive, Pig, Cascading i Crunch, zyskały popularność, ponieważ ręczne programowanie prac w modelu MapReduce jest dość żmudne. Wraz z wprowadzeniem systemu Tez pojawiła się następna korzyść związana z tymi wysokopoziomowymi rozwiązaniami — możliwość zastosowania nowego systemu wykonawczego dla przepływu danych bez konieczności modyfikowania kodu pracy. Także Spark i Flink udostępniają własne wysokopoziomowe interfejsy API do obsługi przepływu danych, często inspirowane biblioteką FlumeJava [34].

Interfejsy API do obsługi przepływów danych zwykle pozwalają zapisywać obliczenia za pomocą komponentów w stylu relacyjnym. Można łączyć zbiory danych na podstawie wartości pola, grupować krotki według kluczy, filtrować dane na podstawie warunku lub agregować krotki za pomocą zliczania, sumowania albo innych funkcji. Wewnętrznie te operacje są implementowane przy użyciu różnych algorytmów łączenia i grupowania opisanych wcześniej w rozdziale.

Obok oczywistej zalety, jaką jest mniejsza ilość kodu, interfejsy wysokopoziomowe są interaktywne i pozwalają stopniowo pisać kod analiz w powłoce oraz często uruchamiać go w celu sprawdzenia, jak działa. Ten styl programowania jest bardzo pomocny w trakcie eksplorowania zbiorów danych i eksperymentowania z metodami przetwarzania go. Przypomina też filozofię Uniksa opisaną w punkcie „Filozofia Uniksa”.

Ponadto te wysokopoziomowe interfejsy nie tylko zwiększają wydajność użytkowników systemu, ale też wydajność wykonywania prac w maszynie.

### W kierunku deklaratywnych języków zapytań

Zaletą zapisywania złączeń jako operatorów relacyjnych (w porównaniu z pisaniem kodu wykonującego złączenie) jest to, że platforma może przeanalizować cechy złączanych danych wejściowych i automatycznie zdecydować, który z algorytmów złączania będzie najbardziej odpowiedni. Hive, Spark i Flink obejmują służące do tego oparte na kosztach optymalizatory zapytań, a nawet zmieniają kolejność złączeń, aby zminimalizować ilość stanu pośredniego [66, 77, 78, 79].

Wybór algorytmu złączania może mieć duży wpływ na wydajność pracy wsadowej. Ponadto wygodnie jest nie musieć rozumieć i zapamiętywać wszystkich algorytmów złączania opisanych w tym rozdziale. Jest to możliwe, jeśli złączenia są zapisywane *deklaratywnie*. W aplikacji wystarczy podać,

które złączenia są potrzebne, a optymalizator zapytań zdecyduje, jak najlepiej je wykonać. Wcześniej ten pomysł pojawił się w punkcie „Języki zapytań o dane”.

Jednak pod innymi względami MapReduce i jego późniejsze odpowiedniki oparte na przepływie danych znacznie się różnią od w pełni deklaratywnego modelu zapytań z SQL-a. Model MapReduce został zaprojektowany z użyciem wywołań zwrotnych. Dla każdego rekordu (lub każdej grupy rekordów) wywoływana jest zdefiniowana przez użytkownika funkcja (mapper lub reducer). Ta funkcja może wywoływać dowolny kod, aby uzyskać dane wyjściowe. Zaletą tego podejścia jest to, że możesz korzystać z dużego ekosystemu istniejących bibliotek do wykonywania zadań takich jak parsowanie, analiza języka naturalnego, analiza obrazów i uruchamiania algorytmów numerycznych lub statystycznych.

Możliwość łatwego uruchamiania dowolnego kodu to coś, co przez długi czas odróżniało systemy przetwarzania wsadowego powiązane z modelem MapReduce od baz MPP (zob. punkt „Porównanie Hadoopa z bazami rozproszonymi”). Choć bazy umożliwiają pisanie funkcji zdefiniowanych przez użytkownika, służące do tego mechanizmy są często niewygodne w użyciu oraz słabo zintegrowane z menedżerami pakietów i systemami zarządzania zależnościami (takimi jak Maven w Javie, npm w JavaScriptcie i Rubygems w języku Ruby) powszechnie stosowanymi w większości języków programowania.

Jednak twórcy systemów przepływu danych odkryli, że stosowanie mechanizmów deklaratywnych ma zalety także w obszarach innych niż złączenia. Na przykład jeśli wywoływana zwrotnie funkcja obejmuje tylko prosty warunek filtrowania lub pobiera wybrane pola z rekordu, wywoływanie jej dla każdego rekordu powoduje znaczne koszty czasu procesora. Gdy takie proste operacje filtrowania i mapowania są zapisywane deklaratywnie, optymalizator zapytań może wykorzystać układ kolumnowy (zob. punkt „Bazy kolumnowe”) i wczytywać z dysku tylko potrzebne kolumny. Hive, Spark DataFrames i Impala korzystają też z wykonywania wektorowego (zob. punkt „Szybkość przenoszenia danych do pamięci i przetwarzanie wektorowe”). Polega to na iteracyjnym przetwarzaniu danych w krótkiej pętli wewnętrznej dogodnej dla pamięci podręcznej procesora i unikaniu wywołań funkcji. Na potrzeby tych pętli wewnętrznych Spark generuje kod bajtowy dla maszyn JVM [79], a Impala używa kompilatora LLVM do generowania kodu natywnego [41].

Dzięki dodaniu aspektów deklaratywnych do wysokopoziomowych interfejsów API i dzięki stosowaniu optymalizatorów zapytań, które potrafią wykorzystać te aspekty w trakcie pracy, platformy przetwarzania wsadowego zaczynają wyglądać bardziej jak bazy MPP (i osiągają porównywalną wydajność). Jednocześnie dzięki możliwości uruchamiania dowolnego kodu i wczytywania danych w dowolnych formatach platformy te zapewniają korzyści związane z elastycznością.

## Specjalizacja pod kątem różnych dziedzin

Choć możliwość uruchamiania dowolnego kodu jest przydatna, standardowe wzorce przetwarzania powtarzają się w wielu sytuacjach. Dlatego warto mieć implementacje standardowych komponentów przeznaczone do wielokrotnego użytku. Tradycyjnie bazy MPP miały zaspokajać potrzeby analityków biznesowych i twórców raportów biznesowych. Są to jednak tylko jedne z wielu dziedzin, w których stosuje się przetwarzanie wsadowe.

Inną dziedziną o rosnącym znaczeniu są algorytmy statystyczne i numeryczne, potrzebne w aplikacjach do uczenia maszynowego (np. w systemach klasyfikacji i rekomendacji). Pojawiają się implementacje przeznaczone do wielokrotnego użytku. Na przykład Mahout obejmuje różne algorytmy uczenia maszynowego oparte na technologiach MapReduce, Spark i Flink, natomiast MADlib udostępnia podobne mechanizmy w relacyjnej bazie MPP (Apache HAWQ) [54].

Przydatne są też algorytmy przestrzenne takie jak *k-najbliższych sąsiadów* [80], które szukają elementów bliskich danemu w przestrzeni wielowymiarowej (jest to jeden ze sposobów wyszukiwania na podstawie podobieństwa). W algorytmach analizy genomu (które mają znajdować podobne, ale nieidentyczne łańcuchy znaków) istotne jest wyszukiwanie przybliżone [81].

Systemy przetwarzania wsadowego są używane do rozproszonego wykonywania algorytmów z coraz liczniejszych dziedzin. Wraz z tym, jak systemy przetwarzania wsadowego zyskują wbudowane funkcje i wysokopoziomowe deklaratywne operatory, a w bazach MPP zwiększają się możliwości programistyczne i elastyczność, oba te rozwiązania zaczynają się upodabniać do siebie. Ostatecznie i jedno, i drugie są systemami składowania i przetwarzania danych.

## Podsumowanie

W tym rozdziale omówiono zagadnienie przetwarzania wsadowego. Najpierw opisano narzędzia uniksowe takie jak `awk`, `grep` i `sort`. Zobaczyłeś, w jaki sposób filozofia projektowa związana z tymi narzędziami znajduje odzwierciedlenie w MapReduce i nowszych systemach przepływu danych. Niektóre z tych zasad projektowych to niezmiennosc danych wejściowych, używanie wyjść jako wejść innego (na razie nieznanego) programu i rozwiązywanie złożonych problemów przez łączenie prostych narzędzi, które „dobrze wykonują jedno zadanie”.

W świecie Uniksa jednolity interfejs, który umożliwia łączenie jednego programu z innym, ma postać plików i potoków. W MapReduce tym interfejsem jest rozproszony system plików. Dowiedziałeś się, że systemy przepływu danych używają własnych, podobnych do potoków mechanizmów przesyłania danych, aby uniknąć materializowania stanu pośredniego w rozproszonym systemie plików. Jednak początkowe wejście i ostateczne wyjście zadania nadal jest zwykle zapisywane w systemie HDFS.

Dwa podstawowe problemy, jakie trzeba rozwiązać w rozproszonych platformach przetwarzania wsadowego, to:

### *Podział*

W MapReduce dane są rozdzielane między mappery na podstawie wejściowych bloków plików. Dane wyjściowe z mapperów są ponownie dzielone, sortowane i scalane w konfigurowalną liczbę partycji dla reducerów. Celem tego procesu jest umieszczenie wszystkich powiązanych danych (np. wszystkich rekordów z tym samym kluczem) w jednym miejscu.

Systemy przepływu danych nowsze niż MapReduce próbują unikać sortowania, chyba że jest ono konieczne. Oprócz tego dzielą dane w podobny sposób.

## *Odporność na błędy*

MapReduce często zapisuje dane na dysku, dzięki czemu można przywrócić stan po awarii jednego zadania bez ponownego uruchamiania całej pracy. Spowalnia to jednak działanie, gdy awarie nie występują. Systemy przepływu danych w mniejszym stopniu materializują stan pośredni i więcej danych przechowują w pamięci. Oznacza to, że po awarii węzła muszą ponownie wykonywać więcej obliczeń. Operatory deterministyczne zmniejszają ilość danych, jakie trzeba ponownie przetworzyć.

Opisano tu kilka algorytmów złączania z modelu MapReduce. Większość z nich jest wewnętrznie stosowana także w bazach MPP i systemach przepływu danych. Opisane rozwiązania dobrze ilustrują działania algorytmów dla dzielonych danych:

### *Złączanie z sortowaniem przez scalanie*

Wszystkie złączane dane wejściowe przechodzą przez mapper pobierający klucz używany przy złączaniu. Dzięki podziałowi, sortowaniu i scalaniu wszystkie rekordy o tym samym kluczu trafiają do tego samego wywołania reducera. Reducer może następnie zwrócić złączone rekordy.

### *Złączanie z rozsyłaniem i haszowaniem*

Jedno z dwóch wejść jest niewielkie, dlatego nie trzeba go dzielić — można je w całości umieścić w tablicy z haszowaniem. Dlatego można uruchomić mapper dla każdej partycji dużego wejścia, wczytać w każdym mapperze tablicę z haszowaniem obejmująca małe wejście, a następnie skanować duże wejście rekord po rekordzie i sprawdzać rekordy w tablicy z haszowaniem.

### *Złączanie z podziałem i haszowaniem*

Jeśli dwa wejścia są podzielone w ten sam sposób (za pomocą tego samego klucza, tej samej funkcji haszującej i tej samej liczby partycji), podejście wykorzystujące tablicę z haszowaniem można niezależnie zastosować do każdej partycji.

W systemach rozproszonego przetwarzania wsadowego celowo stosowany jest ograniczony model programowania. Przyjmuje się, że wywoływane zwrotnie funkcje (mapperzy i reducery) są bezstanowe i nie powodują zewnętrznie widocznych efektów ubocznych (jedynie generują wyznaczone dane wyjściowe). To ograniczenie umożliwia platformie ukrycie za pomocą abstrakcji niektórych trudnych problemów z obszaru systemów rozproszonych. Po wystąpieniu awarii i problemów z siecią można bezpiecznie ponowić próbę wykonania zadań i odrzucić dane wyjściowe z zadań zakończonych niepowodzeniem. Jeśli kilka zadań dotyczących partycji zostało poprawnie ukończonych, widoczne są dane wyjściowe wygenerowane tylko przez jedno z nich.

Dzięki opisanej platformie w kodzie pracy odpowiedzialnej za przetwarzanie wsadowe nie trzeba implementować mechanizmów zapewniania odporności na błędy. Platforma gwarantuje, że ostateczne dane wyjściowe pracy są takie same jak w sytuacji, gdy błędy nie wystąpiły (choć w rzeczywistości zapewne konieczne było ponowienie różnych zadań). Niezawodność jest tu znacznie wyższa niż w usługach online, które obsługują zapytania od użytkowników i zapisują dane w bazie jako efekt uboczny przetwarzania zapytania.

Cechą wyróżniającą pracy wykonującej przetwarzanie wsadowe jest to, że wczytuje dane wejściowe i generuje dane wyjściowe bez modyfikowania tych pierwszych. Oznacza to, że dane wyjściowe są pochodne od wejściowych. Bardzo istotne jest to, że dane wejściowe są *ograniczone*. Mają znany

stały rozmiar (np. jest to zestaw plików dziennika z określonego czasu lub snapshot zawartości bazy). Z powodu tego ograniczenia praca wie, kiedy zakończyła wczytywanie całego wejścia, dlatego ostatecznie kończy tę operację po jej wykonaniu.

W następnym rozdziale opisano przetwarzanie strumieniowe, w którym wejście jest *nieograniczone*. Też wykonywana jest wtedy praca, jednak jej wejścia to niekończące się strumienie danych. W owym modelu praca nigdy się nie kończy, ponieważ w dowolnym momencie mogą się pojawić dodatkowe dane. Zobaczysz, że pod niektórymi względami przetwarzanie strumieniowe i wsadowe są podobne do siebie. Jednak założenie o nieograniczoności strumieni znacznie zmienia sposób budowania systemów.

## Literatura cytowana

[1] Jeffrey Dean i Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, w: „6th USENIX Symposium on Operating System Design and Implementation” (OSDI), grudzień 2004 (<https://research.google.com/archive/mapreduce.html>).

[2] Joel Spolsky, *The Perils of JavaSchools*, [joelonsoftware.com](http://joelonsoftware.com), 25 grudnia 2005 (<https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/>).

[3] Shivnath Babu i Herodotos Herodotou, *Massively Parallel Databases and MapReduce Systems*, „Foundations and Trends in Databases”, rocznik 5, nr 1, s. 1 – 104, listopad 2013 (<https://www.microsoft.com/en-us/research/publication/massively-parallel-databases-and-mapreduce-systems/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F206464%2Fdb-mr-survey-final.pdf>; <http://mr.crossref.org/iPage?doi=10.1561%2F19000000036>).

[4] David J. DeWitt i Michael Stonebraker, *MapReduce: A Major Step Backwards*, pierwotnie opublikowane w: [databasecolumn.vertica.com](http://databasecolumn.vertica.com), 17 stycznia 2008 ([https://homes.cs.washington.edu/~billhowe/mapreduce\\_a\\_major\\_step\\_backwards.html](https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html)).

[5] Henry Robinson, *The Elephant Was a Trojan Horse: On the Death of MapReduce at Google*, [the-paper-trail.org](http://the-paper-trail.org), 25 czerwca 2014 (<http://the-paper-trail.org/blog/the-elephant-was-a-trojan-horse-on-the-death-of-map-reduce-at-google/>).

[6] *The Hollerith Machine*, United States Census Bureau, [census.gov](http://census.gov) ([https://www.census.gov/history/www/innovations/technology/the\\_hollerith\\_tabulator.html](https://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html)).

[7] IBM 82, 83, and 84 Sorters Reference Manual, wydanie A24-1034-1, International Business Machines Corporation, lipiec 1962 ([http://www.textfiles.com/bitsavers/pdf/ibm/punchedCard/Sorter/A24-1034-1\\_82-83-84\\_sorters.pdf](http://www.textfiles.com/bitsavers/pdf/ibm/punchedCard/Sorter/A24-1034-1_82-83-84_sorters.pdf)).

[8] Adam Drake, *Command-Line Tools Can Be 235x Faster than Your Hadoop Cluster*, [aadrake.com](http://aadrake.com), 25 stycznia 2014 (<https://aadrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html>).

[9] GNU Coreutils 8.23 Documentation, Free Software Foundation, Inc., 2014 ([http://www.gnu.org/software/coreutils/manual/html\\_node/index.html](http://www.gnu.org/software/coreutils/manual/html_node/index.html)).

- [10] Martin Kleppmann, *Kafka, Samza, and the Unix Philosophy of Distributed Data*, martin.kleppmann.com, 5 sierpnia 2015 (<http://martin.kleppmann.com/2015/08/05/kafka-samza-unix-philosophy-distributed-data.html>).
- [11] Doug McIlroy, wewnętrzna notatka z Bell Labs, październik 1964; przytoczone w: Dennis M. Ritchie, *Advice from Doug McIlroy*, cm.bell-labs.com (<https://www.bell-labs.com/usr/dmr/www/mdmpipe.html>).
- [12] M. Doug McIlroy, Elliot N. Pinson i Berkley A. Tague, *UNIX Time-Sharing System: Foreword*, „The Bell System Technical Journal”, rocznik 57, nr 6, s. 1899 – 1904, lipiec 1978 (<https://archive.org/details/bstj57-6-1899>).
- [13] Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley, 2003, ISBN: 978-0-13-142901-7 (<http://www.catb.org/~esr/writings/taoup/html/>).
- [14] Ronald Duncan, *Text File Formats — ASCII Delimited Text — Not CSV or TAB Delimited Text*, ronaldlduncan.wordpress.com, 31 października 2009 (<https://ronaldlduncan.wordpress.com/2009/10/31/text-file-formats-ascii-delimited-text-not-csv-or-tab-delimited-text/>).
- [15] Alan Kay, *Is „Software Engineering” an Oxymoron?*, tinlizzie.org (<http://tinlizzie.org/~takashi/IsSoftwareEngineeringAnOxymoron.pdf>).
- [16] Martin Fowler, *InversionOfControl*, martinfowler.com, 26 czerwca 2005 (<https://martinfowler.com/bliki/InversionOfControl.html>).
- [17] Daniel J. Bernstein, *Two File Descriptors for Sockets*, cr.yp.to (<http://cr.yp.to/tcpip/twofd.html>).
- [18] Rob Pike i Dennis M. Ritchie, *The Styx Architecture for Distributed Systems*, Bell Labs Technical Journal, rocznik 4, nr 2, s. 146 – 152, kwiecień 1999 ([http://doc.cat-v.org/inferno/4th\\_edition/styx](http://doc.cat-v.org/inferno/4th_edition/styx)).
- [19] Sanjay Ghemawat, Howard Gobioff i Shun-Tak Leung, *The Google File System*, w: „19th ACM Symposium on Operating Systems Principles” (SOSP), październik 2003 (<https://static.googleusercontent.com/media/research.google.com/pl/archive/gfs-sosp2003.pdf>; <https://dl.acm.org/citation.cfm?doid=945445.945450>).
- [20] Michael Ovsiannikov, Silvius Rus, Damian Reeves i in., *The Quantcast File System*, „Proceedings of the VLDB Endowment”, rocznik 6, nr 11, s. 1092 – 1101, sierpień 2013 (<http://db.disi.unitn.eu/pages/VLDBProgram/pdf/industry/p808-ovsiannikov.pdf>; <https://dl.acm.org/citation.cfm?doid=2536222.2536234>).
- [21] *OpenStack Swift 2.6.1 Developer Documentation*, OpenStack Foundation, docs.openstack.org, marzec 2016 (<https://docs.openstack.org/swift/latest/>).
- [22] Zhe Zhang, Andrew Wang, Kai Zheng i in., *Introduction to HDFS Erasure Coding in Apache Hadoop*, blog.cloudera.com, 23 września 2015 (<http://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/>).
- [23] Peter Cnudde, *Hadoop Turns 10*, yahoohadoop.tumblr.com, 5 lutego 2016 (<http://yahoohadoop.tumblr.com/post/138739227316/hadoop-turns-10>).

- [24] Eric Baldeschwieler, *Thinking About the HDFS vs. Other Storage Technologies*, hortonworks.com, 25 lipca 2012 (<https://hortonworks.com/blog/thinking-about-the-hdfs-vs-other-storage-technologies/>).
- [25] Brendan Gregg, *Manta: Unix Meets Map Reduce*, dtrace.org, 25 czerwca 2013 (<http://dtrace.org/blogs/brendan/2013/06/25/manta-unix-meets-map-reduce/>).
- [26] Tom White, *Hadoop: The Definitive Guide*, wydanie czwarte, O'Reilly Media, 2015, ISBN: 978-1-491-90163-2.
- [27] Jim N. Gray, *Distributed Computing Economics*, Microsoft Research Tech Report MSR-TR-2003-24, marzec 2003 (<https://arxiv.org/ftp/cs/papers/0403/0403019.pdf>).
- [28] Márton Trencsényi, *Luigi vs Airflow vs Pinball*, bytepawn.com, 6 lutego 2016 (<http://bytepawn.com/luigi-airflow-pinball.html>).
- [29] Roshan Sumbaly, Jay Kreps i Sam Shah, *The „Big Data” Ecosystem at LinkedIn*, w: „ACM International Conference on Management of Data” (SIGMOD), lipiec 2013 ([https://www.slideshare.net/s\\_shah/the-big-data-ecosystem-at-linkedin-23512853](https://www.slideshare.net/s_shah/the-big-data-ecosystem-at-linkedin-23512853); <https://dl.acm.org/citation.cfm?doid=2463676.2463707>).
- [30] Alan F. Gates, Olga Natkovich, Shubham Chopra i in., *Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience*, w: „35th International Conference on Very Large Data Bases” (VLDB), sierpień 2009 (<http://www.vldb.org/pvldb/2/vldb09-1074.pdf>).
- [31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain i in., *Hive — A Petabyte Scale Data Warehouse Using Hadoop*, w: „26th IEEE International Conference on Data Engineering” (ICDE), marzec 2010 (<http://i.stanford.edu/~ragho/hive-icde2010.pdf>; <http://ieeexplore.ieee.org/document/5447738/>).
- [32] *Cascading 3.0 User Guide*, Concurrent, Inc., docs.cascading.org, styczeń 2016 (<http://docs.cascading.org/cascading/3.0/userguide/>).
- [33] *Apache Crunch User Guide*, Apache Software Foundation, crunch.apache.org (<https://crunch.apache.org/user-guide.html>).
- [34] Craig Chambers, Ashish Raniwala, Frances Perry i in., *FlumeJava: Easy, Efficient Data-Parallel Pipelines*, w: „31st ACM SIGPLAN Conference on Programming Language Design and Implementation” (PLDI), czerwiec 2010 (<https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/35650.pdf>; <https://dl.acm.org/citation.cfm?doid=1806596.1806638>).
- [35] Jay Kreps, *Why Local State is a Fundamental Primitive in Stream Processing*, oreilly.com, 31 lipca 2014 (<https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing>).
- [36] Martin Kleppmann, *Rethinking Caching in Web Apps*, martin.kleppmann.com, 1 października 2012 (<http://martin.kleppmann.com/2012/10/01/rethinking-caching-in-web-apps.html>).
- [37] Mark Grover, Ted Malaska, Jonathan Seidman i Gwen Shapira, *Hadoop Application Architectures*, O'Reilly Media, 2015, ISBN: 978-1-491-90004-8 (<http://shop.oreilly.com/product/0636920033196.do>).

- [38] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar i in., *Challenges to Adopting Stronger Consistency at Scale*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf>).
- [39] Sriranjana Manjunath, *Skewed Join*, [wiki.apache.org](http://wiki.apache.org), 2009 (<https://wiki.apache.org/pig/PigSkewedJoinSpec>).
- [40] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider i S. Seshadri, *Practical Skew Handling in Parallel Joins*, w: „18th International Conference on Very Large Data Bases” (VLDB), sierpień 1992 (<http://www.vldb.org/conf/1992/P027.PDF>).
- [41] Marcel Kornacker, Alexander Behm, Victor Bittorf i in., *Impala: A Modern, Open-Source SQL Engine for Hadoop*, w: „7th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2015 (<http://pandis.net/resources/cidr15impala.pdf>).
- [42] Matthieu Monsch, *Open-Sourcing PalDB, a Lightweight Companion for Storing Side Data*, [engineering.linkedin.com](http://engineering.linkedin.com), 26 października 2015 (<https://engineering.linkedin.com/blog/2015/10/open-sourcing-paldb--a-lightweight-companion-for-storing-side-da>).
- [43] Daniel Peng i Frank Dabek, *Large-Scale Incremental Processing Using Distributed Transactions and Notifications*, w: „9th USENIX Conference on Operating Systems Design and Implementation” (OSDI), październik 2010 ([https://www.usenix.org/legacy/event/osdi10/tech/full\\_papers/Peng.pdf](https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf)).
- [44] *Cloudera Search User Guide*, Cloudera, Inc., wrzesień 2015 (<http://www.cloudera.com/documentation/cdh/5-1-x/Search/Cloudera-Search-User-Guide/Cloudera-Search-User-Guide.html>).
- [45] Lili Wu, Sam Shah, Sean Choi i in., *The Browsemaps: Collaborative Filtering at LinkedIn*, w: „6th Workshop on Recommender Systems and the Social Web” (RSWeb), październik 2014 ([http://ls13-www.cs.tu-dortmund.de/homepage/rsweb2014/papers/rsweb2014\\_submission\\_3.pdf](http://ls13-www.cs.tu-dortmund.de/homepage/rsweb2014/papers/rsweb2014_submission_3.pdf)).
- [46] Roshan Sumbaly, Jay Kreps, Lei Gao i in., *Serving Large-Scale Batch Computed Data with Project Voldemort*, w: „10th USENIX Conference on File and Storage Technologies” (FAST), luty 2012 ([http://static.usenix.org/events/fast12/tech/full\\_papers/Sumbaly.pdf](http://static.usenix.org/events/fast12/tech/full_papers/Sumbaly.pdf)).
- [47] Varun Sharma, *Open-Sourcing Terrapin: A Serving System for Batch Generated Data*, [engineering.pinterest.com](http://engineering.pinterest.com), 14 września 2015 ([https://medium.com/@Pinterest\\_Engineering/open-sourcing-terrapin-a-serving-system-for-batch-generated-data-7aa2f38c4472](https://medium.com/@Pinterest_Engineering/open-sourcing-terrapin-a-serving-system-for-batch-generated-data-7aa2f38c4472)).
- [48] Nathan Marz, *ElephantDB*, [slideshare.net](http://slideshare.net), 30 maja 2011 (<https://www.slideshare.net/nathanmarz/elephantdb>).
- [49] Jean-Daniel (JD) Cryans, *How-to: Use HBase Bulk Loading, and Why*, [blog.cloudera.com](http://blog.cloudera.com), 27 września 2013 (<http://blog.cloudera.com/blog/2013/09/how-to-use-hbase-bulk-loading-and-why/>).
- [50] Nathan Marz, *How to Beat the CAP Theorem*, [nathanmarz.com](http://nathanmarz.com), 13 października 2011 (<http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>).
- [51] Molly Bartlett Dishman i Martin Fowler, *Agile Architecture*, w: „O’Reilly Software Architecture Conference”, marzec 2015 (<https://conferences.oreilly.com/software-architecture/sa2015/public/schedule/detail/40388>).



- [52] David J. DeWitt i Jim N. Gray, *Parallel Database Systems: The Future of High Performance Database Systems*, „Communications of the ACM”, rocznik 35, nr 6, s. 85 – 98, czerwiec 1992 (<http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/dewittgray92.pdf>; <https://dl.acm.org/citation.cfm?doid=129888.129894>).
- [53] Jay Kreps, *But the multi-tenancy thing is actually really really hard*, seria tweetów, *twitter.com*, 31 października 2014 (<https://twitter.com/jaykreps/status/528235702480142336>).
- [54] Jeffrey Cohen, Brian Dolan, Mark Dunlap i in., *MAD Skills: New Analysis Practices for Big Data*, „Proceedings of the VLDB Endowment”, rocznik 2, nr 2, s. 1481 – 1492, sierpień 2009 (<http://www.vldb.org/pvldb/2/vldb09-219.pdf>; <https://dl.acm.org/citation.cfm?doid=1687553.1687576>).
- [55] Ignacio Terrizzano, Peter Schwarz, Mary Roth i John E. Colinom, *Data Wrangling: The Challenging Journey from the Wild to the Lake*, w: „7th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2015 ([http://cidrdb.org/cidr2015/Papers/CIDR15\\_Paper2.pdf](http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf)).
- [56] Paige Roberts, *To Schema on Read or to Schema on Write, That Is the Hadoop Data Lake Question*, *adaptivesystemsinc.com*, 2 lipca 2015 (<https://adaptivesystemsinc.com/blog/to-schema-on-read-or-to-schema-on-write-that-is-the-hadoop-data-lake-question/>).
- [57] Bobby Johnson i Joseph Adler, *The Sushi Principle: Raw Data Is Better*, w: „Strata+Hadoop World”, luty 2015 (<https://vimeo.com/123985284>).
- [58] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas i in., *Apache Hadoop YARN: Yet Another Resource Negotiator*, w: „4th ACM Symposium on Cloud Computing” (SoCC), październik 2013 (<https://54e57bc8-a-62cb3a1a-s-sites.googlegroups.com/site/2013socc/home/program/a5-vavilapalli.pdf>; <https://dl.acm.org/citation.cfm?doid=2523616.2523633>).
- [59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu i in., *Large-Scale Cluster Management at Google with Borg*, w: „10th European Conference on Computer Systems” (EuroSys), kwiecień 2015 (<https://research.google.com/pubs/pub43438.html>; <https://dl.acm.org/citation.cfm?doid=2741948.2741964>).
- [60] Malte Schwarzkopf, *The Evolution of Cluster Scheduler Architectures*, *firmament.io*, 9 marca 2016 (<http://www.firmament.io/blog/scheduler-architectures.html>).
- [61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das i in., *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, w: „9th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), kwiecień 2012 (<https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>).
- [62] Holden Karau, Andy Konwinski, Patrick Wendell i Matei Zaharia, *Learning Spark*, O'Reilly Media, 2015, ISBN: 978-1-449-35904-1.
- [63] Bikas Saha i Hitesh Shah, *Apache Tez: Accelerating Hadoop Query Processing*, w: „Hadoop Summit”, czerwiec 2014 ([https://www.slideshare.net/Hadoop\\_Summit/w-1205phall1saha](https://www.slideshare.net/Hadoop_Summit/w-1205phall1saha)).
- [64] Bikas Saha, Hitesh Shah, Siddharth Seth i in., *Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2015 ([http://home.cse.ust.hk/~weiwa/teaching/Fall15-COMP6611B/reading\\_list/Tez.pdf](http://home.cse.ust.hk/~weiwa/teaching/Fall15-COMP6611B/reading_list/Tez.pdf); <https://dl.acm.org/citation.cfm?doid=2723372.2742790>).

- [65] Kostas Tzoumas, *Apache Flink: API, Runtime, and Project Roadmap*, slideshare.net, 14 stycznia 2015 (<https://www.slideshare.net/KostasTzoumas/apache-flink-api-runtime-and-project-roadmap>).
- [66] Alexander Alexandrov, Rico Bergmann, Stephan Ewen i in., *The Stratosphere Platform for Big Data Analytics*, „The VLDB Journal”, rocznik 23, nr 6, s. 939 – 964, maj 2014 ([https://ssc.io/pdf/2014-VLDBJ\\_Stratosphere\\_Overview.pdf](https://ssc.io/pdf/2014-VLDBJ_Stratosphere_Overview.pdf); <https://link.springer.com/article/10.1007%2Fs00778-014-0357-y>).
- [67] Michael Isard, Mihai Budiu, Yuan Yu i in., *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*, w: „European Conference on Computer Systems” (EuroSys), marzec 2007 (<https://www.microsoft.com/en-us/research/project/dryad/>; <https://dl.acm.org/citation.cfm?doid=1272996.1273005>).
- [68] Daniel Warneke i Odej Kao, *Nephele: Efficient Parallel Data Processing in the Cloud*, w: „2nd Workshop on Many-Task Computing on Grids and Supercomputers” (MTAGS), listopad 2009 ([https://stratosphere2.dima.tu-berlin.de/assets/papers/Nephele\\_09.pdf](https://stratosphere2.dima.tu-berlin.de/assets/papers/Nephele_09.pdf); <https://dl.acm.org/citation.cfm?doid=1646468.1646476>).
- [69] Lawrence Page, Sergey Brin, Rajeev Motwani i Terry Winograd, *The PageRank Citation Ranking: Bringing Order to the Web*, Stanford InfoLab Technical Report 422, 1999 (<http://ilpubs.stanford.edu:8090/422/>).
- [70] Leslie G. Valiant, *A Bridging Model for Parallel Computation*, „Communications of the ACM”, rocznik 33, nr 8, s. 103 – 111, sierpień 1990 (<https://dl.acm.org/citation.cfm?id=79181>).
- [71] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann i Volker Markl, *Spinning Fast Iterative Data Flows*, „Proceedings of the VLDB Endowment”, rocznik 5, nr 11, s. 1268 – 1279, lipiec 2012 ([http://vldb.org/pvldb/vol5/p1268\\_stephanewen\\_vldb2012.pdf](http://vldb.org/pvldb/vol5/p1268_stephanewen_vldb2012.pdf); <https://dl.acm.org/citation.cfm?doid=2350229.2350245>).
- [72] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik i in., *Pregel: A System for Large-Scale Graph Processing*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2010 ([https://kowshik.github.io/JPregel/pregel\\_paper.pdf](https://kowshik.github.io/JPregel/pregel_paper.pdf); <https://dl.acm.org/citation.cfm?doid=1807167.1807184>).
- [73] Frank McSherry, Michael Isard i Derek G. Murray, *Scalability! But at What COST?*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<http://www.frankmcsherry.org/assets/COST.pdf>).
- [74] Ionel Gog, Malte Schwarzkopf, Natacha Crooks i in., *Musketeer: All for One, One for All in Data Processing Systems*, w: „10th European Conference on Computer Systems” (EuroSys), kwiecień 2015 (<http://www.cl.cam.ac.uk/research/srg/netos/camsas/pubs/eurosys15-musketeer.pdf>; <https://dl.acm.org/citation.cfm?doid=2741948.2741968>).
- [75] Aapo Kyrola, Guy Blelloch i Carlos Guestrin, *GraphChi: Large-Scale Graph Computation on Just a PC*, w: „10th USENIX Symposium on Operating Systems Design and Implementation” (OSDI), październik 2012 (<https://www.usenix.org/system/files/conference/osdi12/osdi12-final-126.pdf>).

- [76] Andrew Lenharth, Donald Nguyen i Keshav Pingali, *Parallel Graph Analytics*, „Communications of the ACM”, rocznik 59, nr 5, s. 78 – 87, maj 2016 (<https://cacm.acm.org/magazines/2016/5/201591-parallel-graph-analytics/abstract>; <https://dl.acm.org/citation.cfm?doid=2930840.2901919>).
- [77] Fabian Hüske, *Peeking into Apache Flink's Engine Room*, [flink.apache.org](http://flink.apache.org), 13 marca 2015 (<http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html>).
- [78] Mostafa Mokhtar, *Hive 0.14 Cost Based Optimizer (CBO) Technical Overview*, [hortonworks.com](http://hortonworks.com), 2 marca 2015 (<https://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/>).
- [79] Michael Armbrust, Reynold S Xin, Cheng Lian i in., *Spark SQL: Relational Data Processing in Spark*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2015 ([http://people.csail.mit.edu/matei/papers/2015/sigmod\\_spark\\_sql.pdf](http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf); <https://dl.acm.org/citation.cfm?doid=2723372.2742797>).
- [80] Daniel Blazeovski, *Planting Quadrees for Apache Flink*, [insightdataengineering.com](http://insightdataengineering.com), 25 marca 2016 (<https://blog.insightdatascience.com/planting-quadrees-for-apache-flink-b396ebc80d35>).
- [81] Tom White, *Genome Analysis Toolkit: Now Using Apache Spark for Data Processing*, [blog.cloudera.com](http://blog.cloudera.com), 6 kwietnia 2016 (<http://blog.cloudera.com/blog/2016/04/genome-analysis-toolkit-now-using-apache-spark-for-data-processing/>).



# Przetwarzanie strumieniowe

*Złożony system, który działa, zawsze jest wynikiem ewolucji prostego działającego systemu. Odwrotne twierdzenie też wydaje się prawdziwe: złożony system zaprojektowany od podstaw nigdy nie działa i nie da się sprawić, by zaczął funkcjonować.*

— John Gall, *Systemantics* (1975)

W rozdziale 10. opisano przetwarzanie wsadowe — techniki wczytywania zbioru plików jako danych wejściowych i generowania nowego zestawu plików wyjściowych. Wyjście ma postać *danych pochodnych*. Oznacza to, że zbiór danych można w razie potrzeby odtworzyć, ponownie uruchamiając proces wsadowy. Zobaczyłeś, w jaki sposób to proste, ale wartościowe podejście zastosować do tworzenia indeksów wyszukiwania i systemów rekomendacji, w analityce itd.

Jednak w całym rozdziale 10. obowiązywało ważne założenie, zgodnie z którym dane wejściowe są ograniczone (czyli mają znany i skończony rozmiar), dzięki czemu proces wsadowy wie, kiedy zakończył ich wczytywanie. Na przykład konieczna w platformie MapReduce operacja sortowania musi wczytać całe dane wejściowe przed rozpoczęciem generowania danych wyjściowych. Możliwe, że ostatni rekord wejściowy będzie miał najniższy klucz, dlatego musi być ostatnim rekordem wyjściowym. Tak więc wcześniejze zwracanie danych wyjściowych nie jest możliwe.

W praktyce dane często są nieograniczone, ponieważ napływają stopniowo w czasie. Użytkownicy generowali dane wczoraj i dziś, a jutro utworzą ich jeszcze więcej. Dopóki nie wypadniesz z rynku, ten proces nigdy się nie zakończy. Dlatego zbiór danych nigdy nie będzie „kompletny” [1]. Z tego powodu mechanizmy przetwarzania wsadowego muszą sztucznie dzielić dane na porcje o stałej długości czasu i np. na zakończenie każdego dnia przetwarzać dane z minionego dnia lub na zakończenie każdej godziny przetwarzać dane z upływającej godziny.

Problem z procesami wsadowymi uruchamianymi raz dziennie polega na tym, że dane wejściowe są odzwierciedlane w danych wyjściowych dopiero dzień później. Dla wielu niecierpliwych użytkowników jest to zbyt późno. Aby zmniejszyć opóźnienie, można uruchamiać przetwarzanie części (np. dane z ostatniej sekundy po jej zakończeniu) lub nawet cały czas, rezygnując ze stałych porcji czasu i przetwarzając każde zdarzenie po jego wystąpieniu. Na tej zasadzie odbywa się *przetwarzanie strumieniowe*.

Ogólnie „strumień” oznacza dane, które są stopniowo udostępniane w czasie. Ten mechanizm występuje w wielu miejscach: w `stdin` i `stdout` w Uniksie, w językach programowania (listy leniwe) [2], w interfejsach API systemów plików (np. klasa `FileInputStream` w Javie), w połączeniach TCP, przy przesyłaniu dźwięku i obrazu przez internet itd.

W tym rozdziale *strumienie zdarzeń* są opisane jako mechanizm zarządzania danymi — nieograniczony i stopniowo przetwarzany odpowiednik danych wsadowych omówionych w poprzednim rozdziale. Najpierw pokazano, jak strumienie są reprezentowane, składowane i przesyłane przez sieć. W punkcie „Bazy danych i strumienie” znajdziesz analizę zależności między strumieniami a bazami. Punkt „Przetwarzanie strumieni” to przegląd podejść i narzędzi do stałego przetwarzania takich strumieni oraz sposobów używania tych technik do budowania aplikacji.

## Przesyłanie strumieni zdarzeń

W świecie przetwarzania wsadowego wejściami i wyjściami pracy są pliki (np. w rozproszonym systemie plików). Jak wygląda ich odpowiednik w strumieniach?

Gdy wejście to plik (sekwencja bajtów), pierwszym krokiem w przetwarzaniu jest zwykle parsowanie go do postaci sekwencji rekordów. W przetwarzaniu strumieniowym rekord bywa częściej nazywany *zdarzeniem*, chodzi jednak o to samo: mały, niezależny, niemodyfikowalny obiekt zawierający informacje o tym, co się zdarzyło w jakimś czasie. Zdarzenie obejmuje zwykle znacznik czasu określający, kiedy (według zegara czasu rzeczywistego) dana rzecz się odbyła (zob. punkt „Zegary monotoniczne a zegary czasu rzeczywistego”).

Takim zdarzeniem może być działanie użytkownika, np. wyświetlenie strony lub zakup towaru. Są też zdarzenia generowane przez maszyny — np. okresowy pomiar w czujniku temperatury lub wskazanie obciążenia procesora. W przykładzie z punktu „Przetwarzanie wsadowe z użyciem narzędzi unixowych” zdarzeniu odpowiada każdy wiersz z dziennika serwera WWW.

Zdarzenie można zakodować jako tekstowy łańcuch znaków, w formacie JSON, w formacie binarnym itd. Opisano to w rozdziale 4. Kodowanie pozwala zapisać zdarzenia np. poprzez dołączenie go do pliku, wstawienie do tabeli relacyjnej lub zapis w bazie dokumentowej. Można też przesłać zdarzenie siecią do innego węzła na potrzeby przetwarzania.

W przetwarzaniu wsadowym plik jest zapisywany raz, a następnie może być wczytywany przez wiele prac. W terminologii z obszaru strumieniowania zdarzenie jest generowane raz przez *producenta* (*dostawcę* lub *nadawcę*), a następnie może być przetwarzane przez wielu *konsumentów* (*subskrybentów* lub *odbiorców*) [3]. W systemie plików nazwa pliku identyfikuje zbiór powiązanych rekordów. W systemie strumieniowania powiązane zdarzenia są zwykle grupowane w *temat* lub *strumień*.

Teoretycznie plik lub baza wystarczają do połączenia producentów z konsumentami. Producent zapisuje każde wygenerowane zdarzenie w magazynie danych, a każdy konsument okresowo sprawdza, czy od czasu ostatniego uruchomienia w magazynie pojawiły się zdarzenia. Tak działa proces wsadowy, gdy przetwarza dane z określonego dnia pod jego koniec.

Jednak gdy zbliżasz się do ciągłego przetwarzania z niskim opóźnieniem, odpytywanie (ang. *polling*) staje się kosztowne, jeśli magazyn danych nie jest zaprojektowany pod kątem takiego użytkowania. Im częściej generujesz zapytania, tym mniejszy odsetek żądań zwraca nowe żądania i tym wyższe stają się koszty. Zamiast tego lepiej jest powiadamiać konsumentów o pojawieniu się nowych zdarzeń.

W przeszłości bazy nie obsługiwały zbyt dobrze powiadomień tego rodzaju. Bazy relacyjne często udostępniają *wyzwalacze*, które mogą reagować na zmiany (np. na wstawienie wiersza do tabeli), ale mają poważne ograniczenia i są w pewnym sensie dodatkiem do projektu baz danych [4, 5]. Zamiast tego opracowano wyspecjalizowane narzędzia na potrzeby dostarczania powiadomień o zdarzeniach.

## Systemy obsługi komunikatów

Częstym podejściem do powiadamiania konsumentów o nowych zdarzeniach jest stosowanie *systemu obsługi komunikatów*. Producent przesyła komunikat dotyczący zdarzenia, po czym ten komunikat zostaje przekazany konsumentom. O tych systemach wspomniano już w punkcie „Przepływ danych za pomocą przekazywania komunikatów”. Tutaj omówione są one bardziej szczegółowo.

Prostym sposobem na zaimplementowanie systemu obsługi komunikatów byłby bezpośredni kanał komunikacji taki jak potok Uniksa lub połączenie TCP między producentem a konsumentem. Jednak większość systemów obsługi komunikatów stanowi rozwinięcie tego podstawowego modelu. Potoki Uniksa i połączenia TCP łączą tylko jednego nadawcę z jednym odbiorcą. Z kolei w systemach obsługi komunikatów wiele węzłów producentów może przysyłać komunikaty do tego samego tematu, a wiele węzłów konsumentów może pobierać komunikaty z tego tematu.

W ramach modelu *publikuj-subskrybuj* w różnych systemach stosowane są rozmaite podejścia. Nie istnieje jedno rozwiązanie właściwe w każdej sytuacji. W celu rozróżniania takich systemów warto zadać dwa następujące pytania:

1. *Co się dzieje, jeśli producenci przysyłają komunikaty szybciej, niż konsumenci mogą je przetwarzać?* Są wtedy trzy ogólne możliwości: system może usuwać komunikaty, umieszczać je w kolejce lub stosować *wsteczną propagację obciążenia* (ang. *backpressure*), czyli blokowanie wysyłania dalszych komunikatów przez producenta (inna nazwa to *kontrola przepływu*). Ta technika jest używana np. w potokach Uniksa i połączeniach TCP, gdzie występuje mały bufor, a jego zapelnienie powoduje zablokowanie nadawcy do czasu pobrania danych z bufora przez odbiorcę (zob. punkt „Przeciążenie sieci i kolejki”).

Jeśli komunikaty są buforowane w kolejce, ważne jest, aby zrozumieć, co się dzieje, gdy kolejka stanie się dłuższa. Czy system ulegnie awarii, jeśli kolejka nie będzie mieścić się w pamięci? A może zapisze komunikaty na dysku? Jeżeli komunikaty są umieszczane na dysku, jak dostęp do niego wpływa na wydajność systemu obsługi komunikatów [6]?

2. *Co się dzieje po awarii lub tymczasowej niedostępności węzła? Czy następuje utrata komunikatów?* Podobnie jak w bazach zapewnienie trwałości może wymagać połączenia zapisu na dysku i replikacji (zob. ramkę „Replikacja i trwałość”), co pociąga za sobą koszty. Jeśli możesz sobie pozwolić na to, by czasem utracić komunikaty, prawdopodobnie uzyskasz wyższą przepustowość i niższe opóźnienie, używając tego samego sprzętu.

To, czy utrata komunikatów jest akceptowalna, w dużym stopniu zależy od aplikacji. W przypadku okresowo przesyłanych odczytów z czujnika lub pomiarów utrata pojedynczych punktów danych zapewne nie jest problemem, ponieważ niewiele później i tak przesłana zostanie zaktualizowana wartość. Zauważ przy tym, że po utracie dużej liczby komunikatów może nie być natychmiast oczywiste, że pomiary są nieprawidłowe [7]. Z kolei jeśli zliczasz zdarzenia, ważniejsze jest, by były one dostarczane niezawodnie, ponieważ każda utrata komunikatu oznacza błędne wskazanie licznika.

Dogodną cechą systemów przetwarzania wsadowego opisaną w rozdziale 10. jest to, że zapewniają one silne gwarancje niezawodności. Zadania zakończone niepowodzeniem są automatycznie ponawiane, a częściowe dane wyjściowe z takich zadań są automatycznie usuwane. To oznacza, że dane wyjściowe są takie same jak w sytuacji, gdyby awaria nie wystąpiła. Pomaga to uprościć model programowania. Dalej w rozdziale opisano, jak zapewnić podobne gwarancje w kontekście strumieniowania.

### Bezpośrednie przesyłanie komunikatów od producentów do konsumentów

Liczne systemy obsługi komunikatów używają bezpośredniego połączenia sieciowego między producentami a konsumentami (bez węzłów pośrednich):

- Multicast z użyciem protokołu UDP powszechnie stosuje się w branży finansowej np. dla strumieni z kursami akcji, gdzie liczy się niskie opóźnienie [8]. Choć sam protokół UDP jest zawodny, protokoły z poziomu aplikacji potrafią odzyskać utracone pakiety (producent musi zapamiętywać przesłane pakiety, aby móc na żądanie ponownie je przesłać).
- Biblioteki obsługi komunikatów bez użycia brokera, takie jak ZeroMQ [9] i nanomsg, działają w podobny sposób. Stosują one multicast z użyciem protokołu TCP lub IP do obsługi komunikatów w modelu publikuj-subskrybuj.
- StatsD [10] i Brubeck [7] używają zawodnego protokołu UDP do pobierania pomiarów z wszystkich maszyn w sieci i monitorowania ich. W protokole StatsD pomiary dotyczące licznika są poprawne tylko wtedy, gdy odebrano wszystkie komunikaty. Używanie UDP sprawia, że takie pomiary są w najlepszym przypadku przybliżeniem [11]. Zob. też ramkę „Protokoły TCP i UDP”.
- Jeśli konsument udostępnia usługę w sieci, producenci mogą bezpośrednio zgłaszać żądania HTTP lub RPC (zob. punkt „Przepływ danych z użyciem usług: REST i RPC”), aby przekazywać komunikaty konsumentom. Na tej zasadzie działa mechanizm *webhooks* [12]. To wzorzec polegający na tym, że adres URL wywołania zwrotnego jednej usługi jest rejestrowany w innej usłudze, która po wystąpieniu zdarzenia przesyła żądanie pod ten adres.

Choć systemy z bezpośrednim przesyłaniem komunikatów działają dobrze w sytuacjach, na potrzeby których je zaprojektowano, zwykle wymagają, aby kod aplikacji uwzględniał możliwość utraty komunikatów. Błędy, na jakie aplikacja jest odporna, są stosunkowo nieliczne. Nawet jeśli protokoły wykrywają i ponownie przesyłają pakiety utracone w sieci, zwykle przyjmują, że producenci i konsumenci są stale dostępni.

Jeśli konsument jest niedostępny, może nie otrzymać przesłanych w tym czasie komunikatów. Niektóre protokoły umożliwiają producentowi ponowienie nieudanej próby dostarczenia komunikatu,



jednak to podejście może nie zadziałać, jeśli nastąpi awaria producenta i utrata bufora z komunikatami, które miały zostać ponowione.

## Brokery komunikatów

Często stosowaną alternatywą jest przysyłanie komunikatów z użyciem *brokera komunikatów* (inna nazwa to *kolejka komunikatów*), który przypomina bazę danych zoptymalizowanych pod kątem obsługi strumieni komunikatów [13]. Broker działa jako serwer, a producenci i konsumenci łączą się z nim jako klienci. Producenci zapisują komunikaty w brokerze, a konsumenci przyjmują je, wczytując z brokera.

Dzięki scentralizowaniu danych w brokerze takie systemy mają większą odporność na pojawianie się i znikanie klientów (z powodu łączenia się, rozłączania i awarii). Za trwałość danych odpowiada wtedy broker. Niektóre brokery przechowują komunikaty tylko w pamięci, natomiast inne (w zależności od konfiguracji) zapisują dane na dysku, aby uniknąć ich utraty wskutek awarii brokera. Gdy konsumenci działają powoli, brokery zwykle obsługują nieograniczone kolejki (zamiast usuwać komunikaty lub stosować kontrolę przepływu), choć może to zależeć od konfiguracji.

Konsekwencją stosowania kolejek jest to, że konsumenci działają zwykle *asynchronicznie*. Gdy producent wysłał komunikat, zwykle musi tylko poczekać na potwierdzenie umieszczenia go w buforze przez brokera. Nie oczekuje na przetworzenie komunikatu przez konsumentów. Komunikaty są dostarczane konsumentom w nieokreślonym momencie w przyszłości. Często dzieje się to w ułamku sekundy, ale nieraz znacznie później (jeśli w kolejce znajdują się zaległe komunikaty).

## Porównanie brokerów komunikatów z bazami danych

Niektóre brokery komunikatów mogą nawet działać w ramach protokołów zatwierdzania dwuetapowego dla transakcji XA lub JTA (zob. punkt „Transakcje rozproszone w praktyce”). To upodabnia je do baz, choć występują istotne praktyczne różnice między brokerami komunikatów a bazami:

- Bazy zwykle przechowują dane do momentu ich bezpośredniego usunięcia, natomiast większość brokerów komunikatów automatycznie usuwa komunikat po jego udanym dostarczeniu do konsumentów. Takie brokery nie nadają się do długoterminowego przechowywania danych.
- Ponieważ brokery szybko usuwają komunikaty, zwykle przyjmuje się, że ich zbiór roboczy jest stosunkowo niewielki (czyli że kolejki są krótkie). Jeśli broker musi umieścić w buforze dużą liczbę komunikatów (możliwe, że zapisując je na dysku, gdy nie mieszczą się w pamięci), ponieważ konsumenci działają powoli, przetwarzanie poszczególnych komunikatów trwa dłużej, a ogólna przepustowość systemu może spaść [6].
- Bazy często udostępniają indeksy pomocnicze i różne sposoby przeszukiwania danych, a brokery komunikatów nieraz obsługują sposoby subskrypcji podzbioru tematów pasujących do podanego wzorca. Są to różne mechanizmy, jednak oba pozwalają klientom wybrać porcję zwracanych danych.
- Wyniki zapytań do bazy są zwykle oparte na snapshotcie danych z określonego momentu. Jeśli później inny klient zapisze w bazie coś, co zmieni wynik zapytania, pierwszy klient nie dowie się, że pobrany wynik jest nieaktualny (chyba że ponowi zapytanie lub pobierze informacje o zmianach). Natomiast brokery komunikatów nie obsługują dowolnych zapytań, ale powiadamiają klienty o zmianach w danych (np. o pojawieniu się nowych komunikatów).

Jest to tradycyjny obraz brokerów komunikatów ujęty w standardach takich jak JMS [14] i AMQP [15] oraz zaimplementowany w oprogramowaniu takim jak RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus i Google Cloud Pub/Sub [16].

### Wielu konsumentów

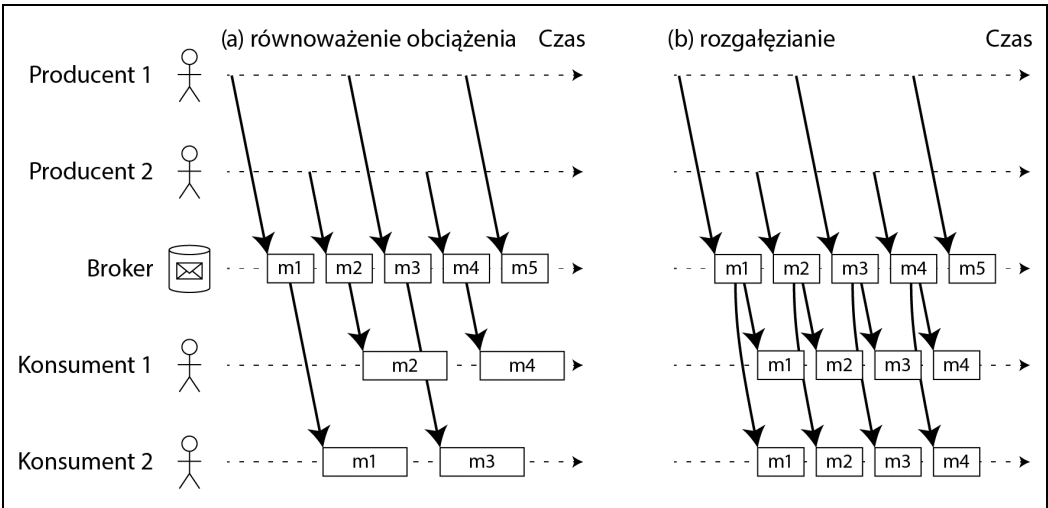
Gdy wielu konsumentów wczytuje komunikaty z jednego tematu, stosowane są dwa główne wzorce obsługi komunikatów (zob. rysunek 11.1):

#### Równoważenie obciążenia

Każdy komunikat jest dostarczany do *jednego* z konsumentów, dzięki czemu przetwarzanie komunikatów z tematu rozdziela się między konsumentów. Broker może dowolnie przypisywać komunikaty do konsumentów. Ten wzorzec się przydaje, gdy przetwarzanie komunikatów jest kosztowne i chcesz mieć możliwość dodania konsumentów na potrzeby przetwarzania równoległego. W AMQP możesz zaimplementować równoważenie obciążenia, stosując kilka klientów do pobierania danych z tej samej kolejki. W JMS ta technika to *subskrypcja wspólna* (ang. *shared subscription*).

#### Rozgałęzianie

Każdy komunikat jest dostarczany do *wszystkich* konsumentów. Rozgałęzianie umożliwia paru niezależnym konsumentom „dostrojenie się” (bez wpływania na siebie) do emitowanych komunikatów. Jest to występujący w kontekście strumieniowania odpowiednik używania kilku różnych prac wsadowych wczytujących ten sam plik wejściowy. Ten mechanizm jest dostępny dzięki subskrypcji tematów w JMS i wiązaniu źródeł danych (ang. *exchange bindings*) w AMQP.



Rysunek 11.1. (a) Równoważenie obciążenia: podział pracy związanej z obsługą tematu między konsumentów. (b) Rozgałęzianie: każdy komunikat jest dostarczany do wielu konsumentów

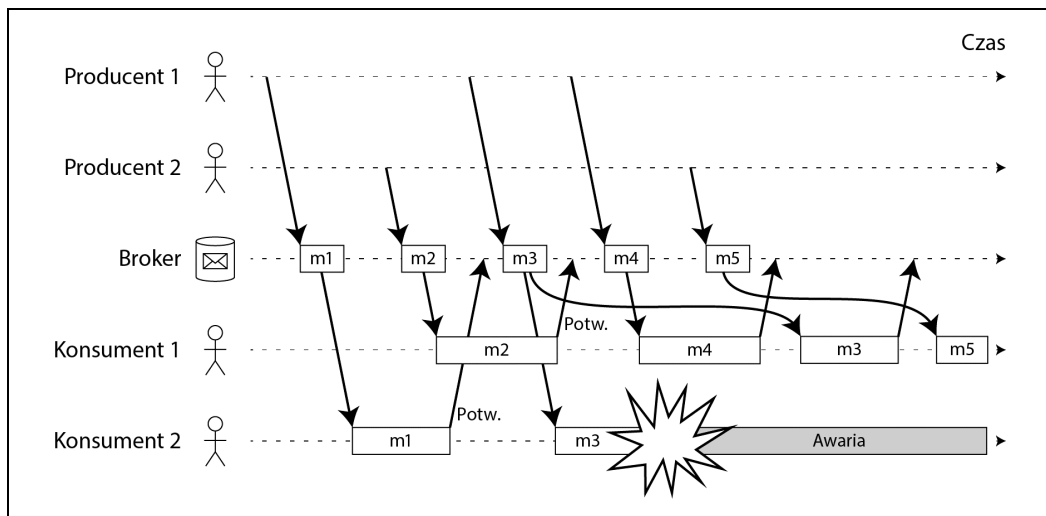
Te dwa wzorce można ze sobą połączyć. Na przykład dwie różne grupy konsumentów mogą subskrybować temat w taki sposób, że każda grupa jako całość otrzymuje wszystkie komunikaty, jednak w ramach każdej grupy poszczególne komunikaty trafiają do tylko jednego węzła.

## Potwierdzenia i ponowne dostarczanie

Konsument może w dowolnym momencie ulec awarii. Dlatego możliwe jest, że broker dostarczy komunikat do konsumenta, ale ten nigdy nie przetworzy danych lub przed wystąpieniem awarii przetworzy je częściowo. Aby zapobiec utracie komunikatu, broker komunikatów stosuje *potwierdzenia*. Klient musi bezpośrednio poinformować brokera o zakończeniu przetwarzania komunikatu, tak by broker mógł usunąć ten komunikat z kolejki.

Jeśli połączenie z klientem zostanie zamknięte lub nastąpi przekroczenie limitu czasu, a broker nie otrzyma potwierdzenia, należy przyjąć, że komunikat nie został przetworzony, i ponownie dostarczyć go do innego konsumenta. Zauważ, że możliwe jest, iż komunikat *został* w pełni przetworzony, ale potwierdzenie zaginęło w sieci. Obsługa takiej sytuacji wymaga protokołu atomowego zatwierdzania opisanego w punkcie „Transakcje rozproszone w praktyce”.

W połączeniu z równoważeniem obciążenia ponawianie dostarczania ma ciekawy wpływ na kolejność komunikatów. Na rysunku 11.2 konsumenci zwykle przetwarzają komunikaty zgodnie z kolejnością ich przesyłania przez producentów. Jednak konsument 2 ulega awarii w trakcie przetwarzania komunikatu *m3*, a w tym samym czasie konsument 1 przetwarza komunikat *m4*. Niepotwierdzony komunikat *m3* jest następnie ponownie dostarczany (do klienta 1). W efekcie klient 1 przetwarza komunikaty w kolejności *m4*, *m3*, *m5*. Tak więc komunikaty *m3* i *m4* nie są dostarczane w tej samej kolejności, w jakiej zostały przesłane przez producenta 1.



Rysunek 11.2. Konsument 2 ulega awarii, gdy przetwarza komunikat *m3*. Dlatego ten komunikat jest później dostarczany ponownie (do konsumenta 1)

Nawet gdy broker komunikatów próbuje zachować ich kolejność (co jest wymagane w standardach JMS i AMQP), połączenie równoważenia obciążenia z ponawianiem dostarczania prowadzi do zmiany kolejności komunikatów. Aby uniknąć tego problemu, można stosować odrębne kolejki dla poszczególnych konsumentów (czyli zrezygnować z równoważenia obciążenia). Zmiana kolejności komunikatów nie stanowi problemu, jeśli są one w pełni niezależne od siebie. Może to jednak mieć znaczenie, jeżeli występują związki przyczynowe między komunikatami (opisano to dalej w rozdziale).

## Podział dzienników na partycje

Przesyłanie pakietu w sieci lub kierowanie żądań do usługi sieciowej to zwykle tymczasowe operacje, które nie pozostawiają trwałego śladu. Choć można je zapisać (za pomocą przechwytywania i rejestrowania pakietów), nie zakłada się, że będą długo dostępne. Nawet broker komunikatów, który trwale zapisuje komunikaty na dysku, szybko je usuwa po dostarczeniu do konsumentów. Dzieje się tak, ponieważ brokery są budowane na potrzeby tymczasowych komunikatów.

W bazach i systemach plików przyjmuje się odwrotne podejście. Oczekuje się, że wszystko, co jest zapisywane w bazie lub pliku, będzie trwałe — przynajmniej do czasu bezpośredniego usunięcia danych.

Ta różnica w podejściu ma duży wpływ na generowanie danych pochodnych. Ważną cechą procesów wsadowych, opisaną w rozdziale 10., jest możliwość wielokrotnego ich uruchamiania i eksperymentowania z krokami przetwarzania bez ryzyka uszkodzenia danych wejściowych (ponieważ są one przeznaczone tylko do odczytu). Przy przysyłaniu komunikatów w modelu typowym dla AMQP i JMS jest inaczej. Otrzymanie komunikatu ma nieodwracalne skutki, jeśli potwierdzenie powoduje usunięcie go z brokera. Nie można więc ponownie uruchomić tego samego konsumenta i oczekiwać tych samych wyników.

Jeśli dodajesz nowego konsumenta do systemu obsługi komunikatów, zwykle zaczyna on otrzymywać komunikaty przesłane dopiero po momencie jego zarejestrowania. Wszystkie wcześniejsze komunikaty są już usunięte i nie można ich odzyskać. Inaczej jest z plikami i bazami, gdzie można dodać nowego klienta w każdym momencie i czytywać w nim dane zapisane w dowolnym czasie w przeszłości (o ile dane nie zostały bezpośrednio nadpisane lub usunięte przez aplikację).

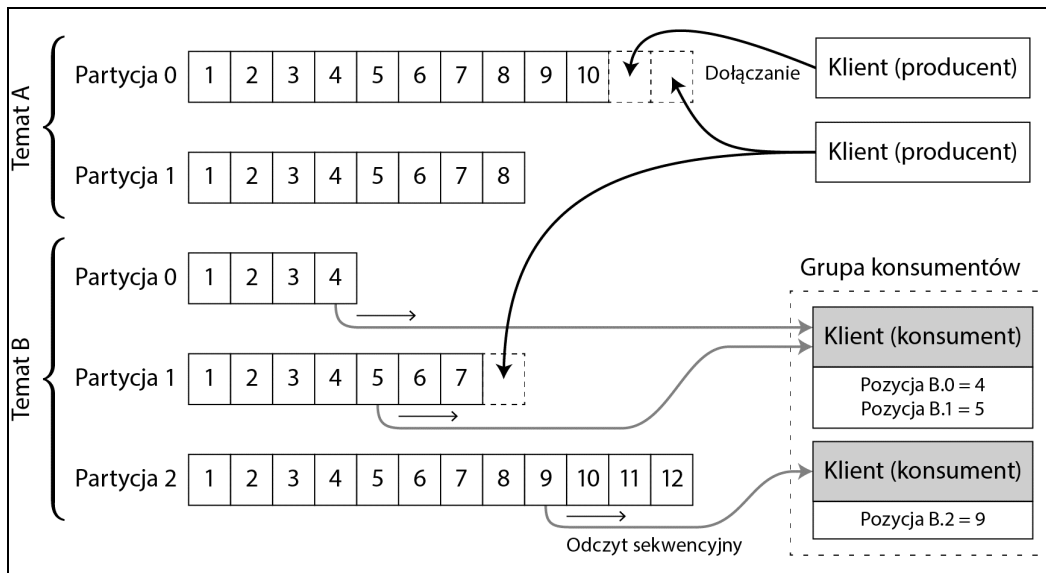
Dlaczego nie utworzyć hybrydowego rozwiązania łączącego trwałą pamięć typową dla baz z powiadomieniami o niskim opóźnieniu z systemów obsługi komunikatów? Na tej zasadzie działają *brokery komunikatów oparte na dziennikach*.

### Używanie dzienników do przechowywania komunikatów

Dziennik to sekwencja rekordów na dysku służąca tylko do dołączania danych. Wcześniej opisano dzienniki w kontekście systemów składowania o strukturze dziennika i dzienników z zapisem wyprzedzającym (rozdział 3.) oraz w związku z replikacją (rozdział 5.).

Tę samą strukturę można też wykorzystać do implementacji brokera komunikatów. Producent wysyła komunikat, dołączając go do dziennika, a konsument pobiera komunikaty, wczytując sekwencyjnie dziennik. Gdy konsument dojdzie do końca dziennika, czeka na powiadomienie o dołączeniu nowego komunikatu. Tak działa uniksowe narzędzie `tail -f` obserwujące plik pod kątem dołączania do niego danych.

Aby umożliwić skalowanie i uzyskanie przepustowości wyższej, niż jest to w stanie zapewnić pojedynczy dysk, dziennik można *podzielić na partycje* (w sensie opisanym w rozdziale 6.). Poszczególne partycje da się umieścić na różnych maszynach, dzięki czemu każda partycja działa jak odrębny dziennik, który można wczytywać i zapisywać niezależnie od pozostałych. Nie ma wtedy przeszkód, by temat zdefiniować jako grupę partycji zawierających komunikaty tego samego typu. To podejście pokazano na rysunku 11.3.



Rysunek 11.3. Producenci przesyłają komunikaty, dołączając je do pliku partycji (tematu). Konsumenty sekwencyjnie wczytują te pliki

W każdej partycji broker przypisuje wszystkim komunikatom monotonicznie rosnące numery porządkowe (*pozycje*). Na rysunku 11.3 pozycjami są numery w polach. Używanie takiej sekwencji numerów jest uzasadnione, ponieważ partycja służy tylko do dołączania danych. Dlatego komunikaty w partycji mają uporządkowanie całkowite. Nie ma jednak gwarancji uporządkowania komunikatów z różnych partycji.

Apache Kafka [17, 18], Amazon Kinesis Streams [19] i DistributedLog firmy Twitter [20, 21] to działające w ten sposób brokery komunikatów oparte na dzienniku. Google Cloud Pub/Sub ma podobną architekturę, jednak udostępnia interfejs API zgodny z modelem JMS, a nie abstrakcję w postaci dziennika [16]. Choć wymienione brokery komunikatów zapisują wszystkie komunikaty na dysku, potrafią uzyskać przepustowość milionów komunikatów na sekundę (dzięki podziałowi danych między wiele maszyn) i odporność na błędy (dzięki replikacji komunikatów) [22, 23].

### Porównanie dzienników z tradycyjną obsługą komunikatów

Podejście oparte na dziennikach zapewnia prostą obsługę rozgałęzienia komunikatów, ponieważ kilku konsumentów może niezależnie wczytywać dziennik bez wpływania na siebie (odczyt komunikatu nie powoduje usunięcia go z dziennika). Na potrzeby równoważenia obciążenia w grupie konsumentów broker może przypisywać całe partycje do węzłów z grupy konsumentów, zamiast przydzielać poszczególne komunikaty konsumentom.

Każdy klient konsumuje wtedy *wszystkie* komunikaty z partycji, która została mu przydzielona. Gdy konsumentowi przydzielona jest partycja dziennika, zwykle wczytuje on komunikaty z partycji sekwencyjnie w prosty, jednowątkowy sposób. Ta ogólna technika równoważenia obciążenia ma kilka wad:

- Liczba węzłów, między które rozdzielana jest praca przetwarzania tematu, nie może być większa niż liczba partycji dziennika dla danego tematu (ponieważ komunikaty z tej samej partycji są kierowane do tego samego węzła)<sup>1</sup>.
- Jeśli przetwarzanie jednego komunikatu trwa długo, wstrzymuje to przetwarzanie dalszych komunikatów z tej partycji (jest to forma blokowania na czole kolejki; zob. punkt „Opis wydajności”).

Dlatego w sytuacji, gdy przetwarzanie komunikatów może być kosztowne, chcesz równolegle przetwarzać je jeden po drugim, a ich kolejność nie ma dużego znaczenia, zaleca się zastosowanie brokera komunikatów w modelu JMS i AMQP. Z kolei gdy częstotliwość komunikatów jest wysoka, są one szybko przetwarzane i ich kolejność ma znaczenie, bardzo dobrze sprawdza się podejście oparte na dzienniku.

### Pozycje odnotowane przez konsumentów

Sekwencyjne konsumowanie partycji pozwala łatwo stwierdzić, które komunikaty zostały już przetworzone. Wszystkie komunikaty z pozycji niższych niż aktualnie odnotowana przez konsumenta zostały już przetworzone; wszystkie komunikaty z pozycji wyższych nie zostały jeszcze pobrane. Dlatego broker nie musi śledzić potwierdzeń przetworzenia każdego komunikatu. Ma jedynie okresowo rejestrować pozycje odnotowane przez konsumentów. Niższe koszty operacji kontrolnych i możliwość przetwarzania wsadowego oraz potokowego pomagają zwiększyć w tym podejściu przepustowość systemów opartych na dzienniku.

Pozycja bardzo przypomina *numer porządkowy z dziennika*, często stosowany w replikacji baz z jednym liderem opisanej w punkcie „Tworzenie nowych obserwatorów”. W replikacji baz numer porządkowy z dziennika umożliwia obserwatorowi ponowne połączenie się z liderem i wznowienie replikacji bez pomijania zapisów. Ta sama zasada jest stosowana w omawianym rozwiązaniu. Broker komunikatów działa jak lider, a konsument jak obserwator.

Jeśli nastąpi awaria węzła konsumenta, powiązana z nim partycja jest przydzielana innemu węzłowi z danej grupy konsumentów. Ten nowy węzeł rozpoczyna przetwarzanie komunikatów od ostatniej odnotowanej pozycji. Jeśli wcześniejszy konsument przetworzył więcej komunikatów, ale nie odnotował ich pozycji, zostaną one przetworzone ponownie. Dalej w rozdziale opisano sposoby radzenia sobie z tym problemem.

### Wykorzystanie przestrzeni na dysku

Jeśli będziesz tylko dołączać dane do dziennika, ostatecznie zabraknie Ci miejsca na dysku. Aby odzyskać to miejsce, dziennik należy dzielić na segmenty i okresowo usuwać dawne segmenty lub przenosić je do pamięci archiwalnej (bardziej zaawansowany sposób zwalniania miejsca na dysku opisano dalej).

---

<sup>1</sup> Można utworzyć system równoważenia obciążenia, w którym dwóch konsumentów wspólnie przetwarza partycję w taki sposób, że obaj wczytują wszystkie komunikaty, przy czym jeden uwzględnia tylko komunikaty z pozycji parzystych, a drugi tylko z pozycji nieparzystych. Innym rozwiązaniem jest rozdzielenie przetwarzania komunikatów między wątki z puli, co jednak komplikuje zarządzanie pozycjami odnotowywanymi przez klientów. Ogólnie zaleca się przetwarzanie partycji w jednym wątku, a stopień równoległości można zwiększyć, dodając partycje.

To sprawia, że jeśli powolny konsument nie nadąza za szybkością napływania komunikatów i jest tak opóźniony, że odnotowana przez niego pozycja prowadzi do usuniętego segmentu, część komunikatów może zostać utracona. Dziennik udostępnia bufor o ograniczonym rozmiarze i usuwa starsze komunikaty po jego zapelnieniu (to tzw. *bufor cykliczny*). Jednak ponieważ ten bufor znajduje się na dysku, może być pojemny.

Wykonajmy proste obliczenia. W czasie, gdy powstaje ta książka, typowy duży dysk twardy ma pojemność 6 TB i szybkość zapisu sekwencyjnego równą 150 MB/s. Jeśli zapisujesz komunikaty z najwyższą możliwą szybkością, zapelnienie dysku zajmie ok. 11 godzin. Dysk mieści więc bufor komunikatów z 11 godzin i po tym czasie zaczyna zastępować starsze komunikaty. Te zależności nie zmieniają się, nawet jeśli używasz wielu dysków i maszyn. W praktyce systemy rzadko zapisują dane na dysku z pełną szybkością, dlatego dziennik zwykle zapewnia bufor na komunikaty z kilku dni, a nawet tygodni.

Niezależnie od czasu przechowywania komunikatów przepustowość dziennika pozostaje mniej więcej stała, ponieważ każdy komunikat i tak jest zapisywany na dysku [18]. Inaczej wygląda to w systemach obsługi komunikatów, które domyślnie przechowują dane w pamięci i zapisują je na dysku tylko wtedy, gdy kolejka stanie się zbyt długa. Takie systemy działają szybko, gdy kolejki są krótkie, ale stają się znacznie wolniejsze, kiedy zaczynają zapisywać dane na dysku. Dlatego przepustowość zależy w nich od ilości zachowywanej historii.

### **Co się dzieje, gdy konsumenci nie nadążają za producentami?**

Na początku punktu „Systemy obsługi komunikatów” opisano trzy rozwiązania możliwe, gdy konsument nie nadąza za szybkością przesyłania komunikatów przez producentów: usuwanie komunikatów, buforowanie ich lub kontrolowanie przepływu. W tym ujęciu podejście oparte na dziennikach to forma buforowania z dużym, ale stałym buforem (ograniczonym przez miejsce dostępne na dysku).

Jeśli konsument ma tak duże opóźnienie, że powinien wczytać komunikaty starsze niż te zapisane na dysku, nie będzie mógł odczytać części komunikatów. Broker gubi więc starsze komunikaty, które nie mieszczą się już w buforze. Możesz śledzić poziom opóźnienia konsumenta względem początku dziennika, i zgłaszać alarm, gdy to opóźnienie jest zbyt duże. Duża pojemność bufora zapewnia operatorowi wystarczająco dużo czasu, aby naprawić powolnego konsumenta i umożliwić mu nadrobienie zaległości, zanim zacznie pomijać komunikaty.

Nawet jeśli konsument jest zanadto opóźniony i zaczyna pomijać komunikaty, dotyczy to tylko tej jednostki. Nie zakłóca to pracy innych konsumentów. Stanowi to istotną korzyść operacyjną. Możesz eksperymentalnie pobierać dane z dziennika produkcyjnego na potrzeby rozwoju oprogramowania, testów lub debugowania, nie martwiąc się o zakłócenie działania usług produkcyjnych. Gdy konsument zostanie zamknięty lub ulegnie awarii, przestanie przetwarzać zasoby. Jedyną rzeczą, jaka zostanie zachowana, jest pozycja odnotowana przez tego konsumenta.

Ten scenariusz wygląda inaczej niż w tradycyjnych brokerach komunikatów, gdzie trzeba starannie usuwać wszystkie kolejki, których konsumenci przestali działać. W przeciwnym razie te kolejki będą niepotrzebnie akumulować komunikaty i zabierać pamięć konsumentom, którzy wciąż są aktywni.

## Odtwarzanie starszych komunikatów

Wcześniej zauważono, że w brokerach komunikatów w modelach AMQP i JMS przetwarzanie oraz potwierdzanie komunikatów to operacja destrukcyjna, ponieważ powoduje usunięcie komunikatów w brokerze. Z kolei w brokerze opartym na kolejkach pobieranie komunikatów bardziej przypomina odczyt z pliku; jest to operacja tylko do odczytu, która nie modyfikuje dziennika.

Jedyny efekt uboczny przetwarzania (obok danych wyjściowych od konsumenta) stanowi zmiana odnotowanej przez konsumenta pozycji. Jednak ta pozycja jest kontrolowana przez konsumenta, dlatego w razie potrzeby można ją łatwo zmodyfikować. Możesz np. uruchomić nowy egzemplarz konsumenta z pozycjami z wczorajszego dnia i zapisywać dane wyjściowe w innej lokalizacji, aby ponownie przetworzyć komunikaty z ostatniego dnia. Możesz powtarzać to dowolną liczbę razy, modyfikując kod przetwarzający dane.

Ten aspekt sprawia, że przetwarzanie komunikatów z użyciem dziennika bardziej przypomina procesy wsadowe z ostatniego rozdziału, gdzie dane pochodne są wyraźnie oddzielone od danych wejściowych w powtarzalnym procesie ich przekształcania. Ułatwia to eksperymentowanie i przywracanie stanu po usterkach oraz błędach, dlatego opisane podejście to dobre narzędzie do integrowania przepływów danych w organizacji [24].

## Strumienie a bazy danych

Przedstawiono tu porównania między brokerami komunikatów a bazami danych. Choć tradycyjnie te dwa mechanizmy są traktowane jak odrębne kategorie narzędzi, zobaczyłeś, że w brokerach komunikatów opartych na dziennikach wykorzystano pomysły z obszaru baz danych i zastosowano te rozwiązania do komunikatów. Można też pójść w odwrotnym kierunku i wykorzystać w bazach pomysły z obszaru komunikatów i strumieni.

Wcześniej stwierdzono, że zdarzenie to zapis czegoś, co stało się w określonym momencie. Tym czymś mogą być działania użytkowników (np. wprowadzenie zapytania w wyszukiwarce) lub odczyt z czytnika, ale też *zapis w bazie*. Zapis w bazie jest zdarzeniem, które można wykryć, zachować i przetworzyć. Z tego spostrzeżenia wynika, że powiązanie między bazami a strumieniami nie ogranicza się do fizycznego przechowywania dzienników na dysku — jest czymś fundamentalnym.

Dziennik replikacji (zob. punkt „Implementowanie dzienników replikacji”) to strumień zdarzeń zapisu w bazie generowany przez lidera w ramach przetwarzania transakcji. Obserwatorzy uwzględniają ten strumień zapisów we własnej kopii bazy, dlatego uzyskują poprawną kopię tych samych danych. Zdarzenia z dziennika replikacji opisują zmiany zaistniałe w danych.

W punkcie „Rozgłaszanie z uporządkowaniem całkowitym” natrafiłeś na *replikację maszyny stanowej*. Napisano tam, że jeśli każde zdarzenie reprezentuje zapis w bazie, a każda replika przetwarza te same zdarzenia w tej samej kolejności, wszystkie repliki uzyskają ten sam ostateczny stan (przyjmuje się, że przetwarzanie zdarzenia to operacja deterministyczna). Jest to więc następny rodzaj strumienia zdarzeń!

W tym punkcie najpierw opisany został problem pojawiający się w jednorodnych systemach danych. Dalej pokazano, jak go rozwiązać, wykorzystując w bazach pomysły z obszaru strumieni zdarzeń.



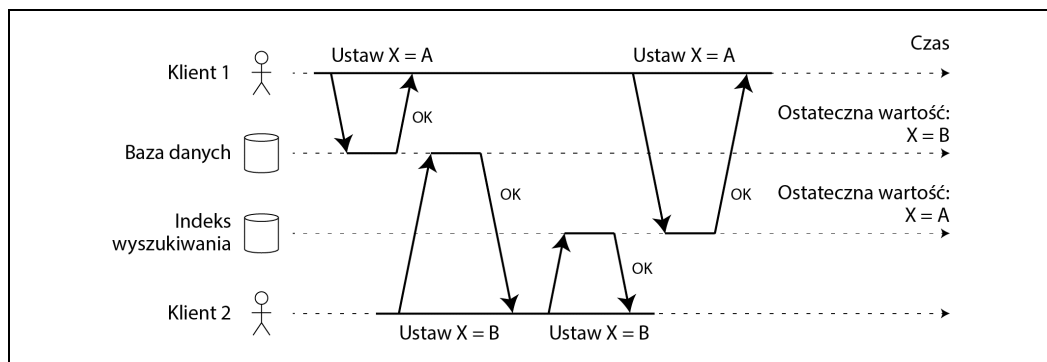
## Utrzymywanie synchronizacji systemów

Z tej książki dowiedziałeś się już, że nie istnieje jeden system, który zaspokaja wszystkie potrzeby z zakresu składowania danych, obsługi zapytań i przetwarzania. W praktyce w większości nieba-  
nalnych aplikacji trzeba łączyć kilka różnych technologii, aby zrealizować stawiane im wymaga-  
nia. Możesz np. zastosować bazę OLTP do obsługi żądań od użytkowników, pamięć podręczną do  
przyspieszenia przetwarzania częstych żądań, indeks pełnotekstowy do obsługi zapytań wymaga-  
jących wyszukiwania i hurtownię danych na potrzeby analityki. Każda z tych technologii wymaga  
własnej kopii danych przechowywanej za pomocą reprezentacji zoptymalizowanej w określonym celu.

Ponieważ te same (lub powiązane) dane pojawiają się w kilku miejscach, trzeba je synchronizować.  
Jeśli element zostanie zaktualizowany w bazie, trzeba go zaktualizować także w pamięci podręcz-  
nej, indeksach wyszukiwania i hurtowni danych. W hurtowniach danych synchronizacja odbywa  
się zwykle za pomocą procesów ETL (zob. punkt „Hurtownie danych”), które często pobierają pełną  
kopię danych, przekształcają ją, a następnie masowo wczytują do hurtowni. Oznacza to zastosowanie  
procesu wsadowego. Podobnie z punktu „Dane wyjściowe ze wsadowych przepływów pracy” dowie-  
działeś się, że indeksy wyszukiwania, systemy rekomendacji i inne systemy z danymi pochodnymi  
też mogą używać procesów wsadowych.

Jeśli okresowe zrzuty pełnej bazy danych zajmują za dużo czasu, czasem stosowane jest inne roz-  
wiązanie — *zapis podwójny*. Polega to na tym, że kod aplikacji po zmodyfikowaniu danych bez-  
pośrednio zapisuje je w każdym systemie. Na przykład najpierw zapisuje je w bazie, następnie  
aktualizuje indeks wyszukiwania, a potem unieważnia dane w pamięci podręcznej (lub nawet wyko-  
nuje wszystkie te operacje jednocześnie).

Jednak z zapisem podwójnym związane są poważne problemy. Jednym z nich jest sytuacja wyścigu  
(rysunek 11.4). W tym przykładzie dwa klienty chcą jednocześnie zaktualizować element X.  
Klient 1 chce przypisać mu wartość A, a klient 2 — B. Oba klienty najpierw zapisują nową wartość  
w bazie, a następnie w indeksie wyszukiwania. Z powodu niefortunnego zbiegu wydarzeń w czasie  
żądania się przeplatają. Baza najpierw widzi, jak klient 1 ustawia wartość na A, a potem jak klient 2  
ustawia ją na B. Dlatego ostateczna wartość w bazie to B. W indeksie wyszukiwania najpierw wi-  
doczny jest zapis klienta 2, a potem zapis klienta 1, dlatego końcowa wartość w indeksie to A.  
Systemy są teraz trwale niespójne ze sobą, choć nie wystąpił błąd.



Rysunek 11.4. W bazie X najpierw jest ustawiane na A, a potem na B. W indeksie wyszukiwania zapisy są wprowadzane w odwrotnej kolejności

Jeśli nie stosujesz dodatkowych mechanizmów wykrywania współbieżności (np. wektorów wersji opisanych w punkcie „Wykrywanie jednoczesnych zapisów”), nawet nie zauważysz, że nastąpił jednoczesny zapis. Jedna wartość po prostu zastąpi drugą.

Inny problem z zapisem podwójnym dotyczy tego, że jeden zapis może się powieść, a inny nie. Jest to w większym stopniu problem z odpornością na błędy niż ze współbieżnością, jednak skutek może być też taki, że systemy stają się niespójne. Zapewnianie, że albo obie operacje zakończą się powodzeniem, albo obie się nie powiedą, to rodzaj problemu zatwierdzania atomowego, którego rozwiązanie jest kosztowne (zob. punkt „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”).

Jeśli używasz tylko jednej bazy z replikacją z jednym liderem, to lider ustala kolejność zapisów. Dlatego dla replik tej bazy zadziała replikacja maszyny stanowej. Jednak na rysunku 11.4 nie występuje jeden lider. Baza może mieć lidera i indeks wyszukiwania może mieć lidera, jednak nie obserwują oni swoich poczynąń, dlatego może wystąpić konflikt (zob. punkt „Replikacja z wieloma liderami”).

Sytuacja byłaby bardziej korzystna, gdyby rzeczywiście występował tylko jeden lider (np. baza), a indeks wyszukiwania byłby obserwatorem. Jednak czy jest to możliwe w praktyce?

## Przechwytywanie zmian w danych

Problem z dziennikami replikacji w większości baz polega na tym, że dzienniki od dawna są uznawane za wewnętrzny szczegół implementacji, a nie za publiczny interfejs API. Klienci powinni kierować zapytania do bazy z użyciem jej modelu danych i języka zapytań, a nie parsować dzienniki replikacji i próbować pobierać z nich dane.

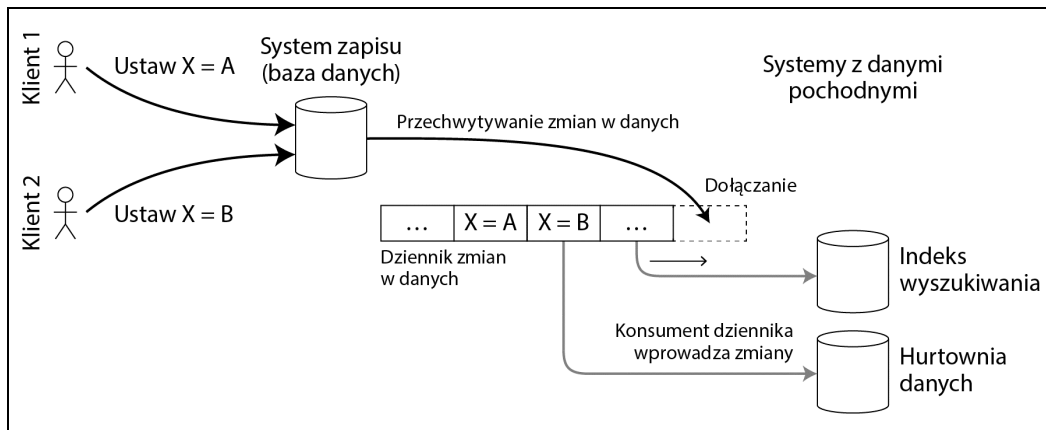
Przez długi czas w wielu bazach nie dokumentowano sposobu pobierania dziennika wprowadzonych zmian. Dlatego trudno było pobrać wszystkie zmiany wprowadzone w bazie i zreplikować je w innej technologii składowania danych (np. w indeksie wyszukiwania, pamięci podręcznej lub hurtowni danych).

Od niedawna nastąpił wzrost zainteresowania techniką *przechwytywania zmian w danych* (ang. *change data capture* — **CDC**). To proces obserwowania wszystkich zmian wprowadzanych w bazie i pobierania ich w formie możliwej do zreplikowania w innych systemach. Technika CDC jest szczególnie interesująca, jeśli zmiany są udostępniane w formie strumienia natychmiast po ich wprowadzaniu.

Możesz np. przechwytywać zmiany z bazy i na bieżąco wprowadzać te same modyfikacje w indeksie wyszukiwania. Jeśli dziennik zmian zostanie uwzględniony w tej samej kolejności, możesz oczekiwać, że dane w indeksie wyszukiwania będą zgodne z bazą. Indeks wyszukiwania i inne systemy z danymi pochodnymi są tylko konsumentami strumienia zmian. Ilustruje to rysunek 11.5.

### Implementowanie przechwytywania zmian w danych

Konsumentów dziennika można nazwać *systemami z danymi pochodnymi*, co opisano we wprowadzeniu do części III. Dane w indeksie wyszukiwania i hurtowni to inne ujęcie danych z systemu zapisu. Przechwytywanie zmian w danych to mechanizm zapewniania, że wszystkie modyfikacje wprowadzone w systemie zapisu są odzwierciedlane także w systemach z danymi pochodnymi. Dzięki temu w tych ostatnich dostępna będzie poprawna kopia danych.



Rysunek 11.5. Pobieranie danych zgodnie z kolejnością ich zapisu w jednej bazie i wprowadzanie zmian w innych systemach w tej samej kolejności

Przechwytywanie zmian w danych powoduje, że jedna baza (ta, z której zmiany są przechwytywane) staje się liderem, a pozostałe systemy są obserwatorami. Broker komunikatów oparty na dzienniku dobrze nadaje się do przesyłania zdarzeń reprezentujących modyfikacje ze źródłowej bazy, ponieważ zachowuje kolejność komunikatów (pozwala to uniknąć problemu zmiany kolejności występującego na rysunku 11.2).

Do zaimplementowania przechwytywania zmian w danych można wykorzystać wyzwalacze bazy danych (zob. punkt „Replikacja oparta na wyzwalaczach”). Należy wtedy zarejestrować wyzwalacze wykrywające wszystkie zmiany w tabelach z danymi i dodające odpowiednie pozycje w dzienniku zmian. Jednak to rozwiązanie jest podatne na błędy i powoduje spadek wydajności. Solidniejszym podejściem może być parsowanie dziennika replikacji, choć ta technika też się wiąże z wyzwaniami (np. z obsługą zmian w schemacie).

Ten pomysł na dużą skalę wykorzystano w narzędziach Databus w serwisie LinkedIn [25], Wormhole w serwisie Facebook [26] i Sherpa w serwisie Yahoo! [27]. Bottled Water to implementacja technologii CDC dla bazy PostgreSQL wykorzystująca interfejs API dekodujący dziennik zapisu z wyprzedzeniem [28]. Narzędzia Maxwell i Debezium wykonują podobne operacje dla MySQL-a za pomocą parsowania dziennika binlog [29, 30, 31], Mongoriver wczytuje dziennik operacji bazy MongoDB [32, 33], a GoldenGate zapewnia zbliżone mechanizmy dla baz Oracle [34, 35].

Przechwytywanie zmian w danych (podobnie jak brokery komunikatów) czasem działa asynchronicznie. System zapisu (baza) nie czeka z zatwierdzeniem zmian na wprowadzenie ich u konsumentów. Ten projekt ma tę zaletę operacyjną, że dodanie powolnego konsumenta nie wpływa w znacznym stopniu na system zapisu. Wadą są problemy związane z opóźnieniem replikacji (zob. punkt „Problemy z opóźnieniem replikacji”).

## Początkowy snapshot

Jeśli masz dziennik wszystkich zmian wprowadzonych w bazie, możesz odtworzyć cały stan bazy, ponownie przetwarzając dziennik. Jednak w wielu sytuacjach przechowywanie w nieskończoność wszystkich modyfikacji wymaga zbyt wiele miejsca na dysku, a odtwarzanie ich trwałoby zbyt długo. Dlatego dzienniki trzeba przycinać.

Na przykład budowanie nowego indeksu pełnotekstowego wymaga pełnej kopii całej bazy. Nie wystarczy zastosować dziennika najnowszych zmian, ponieważ brakuje w nim pozycji, które nie były ostatnio modyfikowane. Dlatego jeśli nie masz pełnej historii dziennika, musisz zacząć od spójnego snapshota, co opisano wcześniej w punkcie „Tworzenie nowych obserwatorów”.

Snapshot bazy musi odpowiadać określonej pozycji z dziennika zmian. Dzięki temu wiadomo, od jakiego punktu zacząć wprowadzać zmiany po przetworzeniu snapshota. Niektóre narzędzia CDC oferują zintegrowaną obsługę snapshotów, natomiast inne wymagają ręcznego zarządzania nimi.

## Kompresja dzienników

Jeśli możesz przechowywać tylko ograniczoną ilość danych w dzienniku, musisz używać snapshota za każdym razem, gdy chcesz dodać nowy system z danymi pochodnymi. Dobrym innym rozwiązaniem jest *kompresja dzienników*.

Kompresja dzienników została wcześniej opisana w punkcie „Indeksy z haszowaniem” w kontekście systemów składowania danych o strukturze dziennika (zob. np. rysunek 3.2). Zasady pracy są tu proste: system składowania danych okresowo wyszukuje w dzienniku rekordy o tym samym kluczu, odrzuca duplikaty i zachowuje tylko najnowszą aktualizację każdego klucza. Ten proces kompresji i scalania działa w tle.

W systemie składowania danych o strukturze dziennika aktualizacja ze specjalną wartością pustą (*nagrobek*) informuje, że klucz został usunięty, i powoduje wykasowanie go w trakcie kompresji. Jednak o ile klucz nie został nadpisany lub usunięty, pozostaje w dzienniku na zawsze. Ilość miejsca na dysku potrzebnego dla tak skompresowanego dziennika zależy tylko od aktualnej zawartości bazy, a nie od łącznej liczby zapisów, jakie w niej wprowadzono. Jeśli ten sam klucz jest często nadpisywany, system odzyskuje pamięć zajmowaną przez jego wcześniejsze wartości i zachowuje tylko najnowszą wartość.

Ten sam pomysł sprawdza się dla brokerów komunikatów opartych na dzienniku i przy przechwytywaniu zmian w danych. Jeśli system CDC został tak skonfigurowany, że dla każdej zmiany zapisywany jest klucz główny, a każda aktualizacja klucza zastępuje jego wcześniejszą wartość, wystarczy zachować tylko najnowszy zapis dotyczący danego klucza.

Teraz gdy zechcesz odtworzyć system z danymi pochodnymi (np. indeks wyszukiwania), możesz uruchomić nowego konsumenta z pozycją 0 ze skompresowanego dziennika i sekwencyjnie przetworzyć wszystkie komunikaty z dziennika. Wiadomo, że dziennik zawiera najnowszą wartość każdego klucza z bazy (i może dodatkowo starsze wartości). Oznacza to, że można uzyskać kompletną kopię zawartości bazy bez konieczności wykonywania innego snapshota źródłowej bazy używanej w procesie CDC.

Mechanizm kompresji dzienników jest obsługiwany w brokerze Apache Kafka. Dalej w rozdziale zobaczysz, że dzięki temu ten broker komunikatów może być stosowany do trwałego przechowywania danych, a nie tylko do obsługi tymczasowych komunikatów.

## Obsługa strumieni zmian za pomocą interfejsu API

Bazy danych coraz częściej udostępniają strumienie zmian w podstawowym interfejsie, a nie w ramach typowego wcześniej dokładania techniki CDC za pomocą wstecznej inżynierii. Na przy-

kład baza RethinkDB umożliwia w zapytaniach subskrypcję powiadomień o zmianach wyników zapytania [36], Firebase [37] i CouchDB [38] obsługują synchronizację danych opartą na kanale zmian, który jest dostępny także dla aplikacji, a Meteor używa dziennika operacji z bazy MongoDB do subskrybowania zmian w danych i aktualizowania interfejsu użytkownika [39].

Baza VoltDB pozwala transakcjom na ciągle eksportowanie strumienia danych z bazy [40]. W relacyjnym modelu danych baza reprezentuje strumień wyjściowy jako tabelę, w której transakcje mogą wstawiać krotki. Nie można jednak kierować zapytań o tę tabelę. Strumień ma wtedy postać dziennika krotek zapisanych przez zatwierdzone transakcje w tej specjalnej tabeli. Kolejność krotek odpowiada porządkowi zatwierdzania transakcji. Zewnętrzni konsumenci mogą asynchronicznie przetwarzać ten dziennik i używać go do aktualizowania systemów z danymi pochodnymi.

Kafka Connect [41] to próba zintegrowania narzędzi CDC dla dużej grupy systemów bazodanowych wykorzystujących Kafkę. Gdy strumień zdarzeń reprezentujących zmiany jest już zapisany w Kafce, można go wykorzystać do aktualizacji systemów z danymi pochodnymi (np. indeksów wyszukiwania), a także przekazać do systemów przetwarzania strumieniowego, co opisano dalej w rozdziale.

## Event sourcing

Występują pewne analogie między opisywanymi tu pomysłami a metodą *event sourcing* — techniką opracowaną przez społeczność **DDD** (ang. *domain-driven design*) [42, 43, 44]. Omówiono tu pokrótce tę metodę, ponieważ obejmuje przydatne i ważne idee związane z systemami strumieniowania.

Event sourcing, podobnie jak przechwytywanie zmian w danych, polega na przechowywaniu wszystkich zmian stanu aplikacji jako dziennika zdarzeń reprezentujących modyfikacje. Największa różnica polega na tym, że w metodzie event sourcing to rozwiązywanie jest stosowane na innym poziomie abstrakcji:

- W CDC aplikacja używa bazy danych, którą może modyfikować, swobodnie aktualizując i usuwając rekordy. Dziennik zmian jest pobierany z bazy niskopoziomowo (np. za pomocą parsowania dziennika replikacji), co gwarantuje, że kolejność zapisów pobranych z bazy odpowiada kolejności ich wprowadzania. Pozwala to uniknąć sytuacji wyścigu widocznej na rysunku 11.4. Aplikacja zapisująca dane w bazie nie musi wiedzieć o stosowaniu CDC.
- W metodzie event sourcing logika aplikacji jest bezpośrednio oparta na niemodyfikowalnych zdarzeniach zapisywanych w dzienniku zdarzeń. W tym podejściu magazyn zdarzeń umożliwia tylko dołączanie danych. Aktualizowanie i usuwanie danych jest odradzane lub zabronione. Zdarzenia są tak projektowane, by odzwierciedlały to, co zdarzyło się w aplikacji, zamiast obrazować niskopoziomowe zmiany stanu.

Event sourcing to dająca duże możliwości technika modelowania danych. Z perspektywy aplikacji bardziej sensowne jest zapisywanie działań użytkowników jako niemodyfikowalnych zdarzeń niż rejestrowanie skutków tych działań w modyfikowalnej bazie. Event sourcing upraszcza późniejsze modyfikowanie aplikacji, pomaga w jej debugowaniu (ponieważ ułatwia zrozumienie, dlaczego coś się stało) i chroni przed błędami (zob. punkt „Zalety niemodyfikowalnych zdarzeń”).

Na przykład zdarzenie „student anulował zapis na kurs” jednoznacznie opisuje cel jednej operacji w neutralny sposób. Natomiast efekty uboczne „jeden wpis został usunięty z tabeli zapisów i jeden powód anulowania został dodany do tabeli z informacjami zwrotnymi od studentów” obejmują wiele założeń dotyczących tego, jak dane mają być później używane. Po dodaniu nowej funkcji aplikacji (np. „miejsce na kursie jest proponowane następnej osobie z listy oczekujących”) metoda event sourcing umożliwia łatwe połączenie nowych efektów ubocznych z istniejącym zdarzeniem.

Event sourcing przypomina model danych kroniki [45]. Występują też podobieństwa między dziennikiem zdarzeń a tabelą faktów ze schematu gwiazdy (zob. „Gwiazdy i płatki śniegu — schematy używane w analityce”).

Opracowane zostały wyspecjalizowane bazy (takie jak Event Store [46]), które zapewniają obsługę aplikacji stosujących metodę event sourcing. Jednak ogólnie ta metoda jest niezależna od narzędzi. Do budowania aplikacji w tym podejściu można wykorzystać także tradycyjną bazę lub brokera komunikatów opartego na dzienniku.

### Uzyskiwanie aktualnego stanu na podstawie dziennika zdarzeń

Dziennik zdarzeń sam w sobie nie jest bardzo przydatny, ponieważ użytkownicy zwykle oczekują, że zobaczą aktualny stan systemu, a nie historię modyfikacji. Na przykład w sklepie internetowym użytkownicy spodziewają się ujrzeć obecną zawartość koszyka, a nie umożliwiającą tylko dołączanie danych listę wszystkich zmian wprowadzonych w tym koszyku.

Dlatego aplikacje używające metody event sourcing muszą pobierać dziennik zdarzeń (reprezentujący dane *zapisane* w systemie) i przekształcać go w stan aplikacji odpowiedni do wyświetlenia użytkownikowi (w sposób, w jaki dane są *wczytywane* z systemu [47]). W tych przekształceniach można się posłużyć dowolną logiką, jednak powinny być one deterministyczne, by można uruchomić je ponownie i na podstawie dziennika zdarzeń uzyskać ten sam stan aplikacji.

Odgrywanie zdarzeń z dziennika (podobnie jak CDC) pozwala odtworzyć aktualny stan systemu. Należy jednak zachować ostrożność przy kompresji dziennika:

- W CDC zdarzenie aktualizacji rekordu zwykle obejmuje całą nową wersję rekordu. Dlatego aktualna wartość dla klucza głównego jest w całości ustalana na podstawie najnowszego zdarzenia dotyczącego tego klucza. W trakcie kompresji dziennika można więc usunąć wcześniejsze zdarzenia związane z tym kluczem.
- W metodzie event sourcing zdarzenia są modelowane na wyższym poziomie. Zdarzenie zwykle reprezentuje cel działań użytkownika, a nie szczegóły aktualizacji stanu wynikającej z tych działań. W tym podejściu późniejsze zdarzenia zwykle nie zastępują wcześniejszych, dlatego wymaga się pełnej historii zdarzeń, aby odtworzyć ostateczny stan. Nie jest więc możliwa standardowa kompresja dziennika.

Aplikacje, w których używa się metody event sourcing, zwykle obejmują mechanizm zapisywania snapshotów bieżącego stanu generowanego na podstawie dziennika zdarzeń. Nie muszą więc wielokrotnie przetwarzać pełnego dziennika. To jednak tylko optymalizacja wydajności mająca przyspieszać odczyty i odzyskiwanie stanu po awariach. Cel jest taki, by system mógł w nieskończoność przechowywać wszystkie nieprzetworzone zdarzenia i w razie potrzeby ponownie przetworzyć kompletny dziennik zdarzeń. To założenie omówiono w punkcie „Ograniczenia niemodyfikowalności”.

## Polecenia i zdarzenia

W metodzie event sourcing obowiązuje filozofia starannego rozróżniania *zdarzeń* od *poleceń* [48]. Gdy nadchodzi żądanie od użytkownika, początkowo jest poleceniem. Na tym etapie przetwarzanie żądania może się jeszcze zakończyć niepowodzeniem (np. z powodu naruszenia warunku zachowania integralności). Aplikacja musi najpierw sprawdzić, czy może wykonać dane polecenie. Jeśli sprawdzanie zakończy się powodzeniem i polecenie zostanie zaakceptowane, staje się zdarzeniem — trwałym i niemodyfikowalnym.

Jeżeli użytkownik spróbuje np. zarejestrować określoną nazwę albo zarezerwować miejsce w samolocie lub teatrze, aplikacja musi najpierw sprawdzić, czy nazwa lub miejsce nie są zajęte (wcześniej ten przykład został opisany w punkcie „Konsensus z zachowaniem odporności na błędy”). Gdy sprawdzanie kończy się powodzeniem, aplikacja może wygenerować zdarzenie oznaczające, że dana nazwa została zarejestrowana przez użytkownika o określonym identyfikatorze lub że konkretne miejsce zostało zarezerwowane dla danego klienta.

W momencie wygenerowania zdarzenia staje się ono *faktem*. Nawet jeśli klient później zdecyduje się zmienić lub anulować rezerwację, pozostaje faktem, że wcześniej zarezerwował określone miejsce. Zmiana lub anulowanie to odrębne, dodawane później zdarzenie.

Konsument strumienia zdarzeń nie może odrzucić zdarzenia. W momencie, w którym konsument otrzymuje zdarzenie, jest już ono niemodyfikowalną częścią dziennika (i możliwe, że zostało zobaczone przez innych konsumentów). Dlatego sprawdzanie poprawności polecenia musi się odbywać synchronicznie, zanim stanie się ono zdarzeniem. Można wykorzystać np. sekwencyjną transakcję, która atomowo sprawdza poprawność polecenia i publikuje zdarzenie.

Inną możliwością jest podział żądania rezerwacji miejsca na dwa zdarzenia — wstępną rezerwację i odrębne zdarzenie potwierdzenia po sprawdzeniu możliwości jej dokonania (tak jak w punkcie „Implementowanie bazy liniowej za pomocą rozgłaszania z uporządkowaniem całkowitym”). Ten podział umożliwia sprawdzanie żądania w asynchronicznym procesie.

## Stan, strumienie i niemodyfikowalność

W rozdziale 10. opisano, że w przetwarzaniu wsadowym korzystna jest niemodyfikowalność plików wejściowych. Pozwala ona uruchamiać eksperymentalne prace na istniejących plikach wejściowych bez obaw o ich uszkodzenie. Zasada niemodyfikowalności sprawia też, że event sourcing i CDC są tak wartościowe.

Zwykle przyjmujemy, że bazy przechowują aktualny stan aplikacji. Ta reprezentacja jest zoptymalizowana pod kątem odczytów i zwykle stanowi najwygodniejszy sposób obsługi zapytań. Naturalne jest to, że stan się zmienia. Dlatego bazy umożliwiają aktualizowanie i usuwanie danych, a także ich wstawianie. Jak się to ma do niemodyfikowalności?

Za każdym razem, gdy używasz modyfikowalnego stanu, jest on wynikiem zdarzeń, które go zmieniły. Na przykład lista aktualnie dostępnych miejsc to wynik przetworzonych rezerwacji, obecny stan konta to efekt uznań i obciążeń, a wykres czasu odpowiedzi serwera WWW to skutek agregacji pojedynczych czasów odpowiedzi dla wszystkich zgłoszonych żądań.

Niezależnie od tego, jak zmienia się stan, zawsze istnieje sekwencja zdarzeń, które doprowadziły do modyfikacji. Nawet gdy operacje są wykonywane i wycofywane, pozostaje faktem, że doszło do zdarzeń. Najważniejsze jest to, że modyfikowalność stanu i umożliwiający tylko dołączanie danych dziennik niemodyfikowalnych zdarzeń nie są ze sobą sprzeczne. Są to dwie strony tej samej monety. Dziennik wszystkich zmian (ang. *changelog*) reprezentuje zmianę stanu w czasie.

Jeśli masz skłonności matematyczne, możesz powiedzieć, że stan aplikacji to coś, co uzyskasz po scałkowaniu strumienia zdarzeń w czasie, a strumień zdarzeń to coś, co otrzymasz w wyniku różniczkowania stanu na podstawie czasu (zob. rysunek 11.6) [49, 50, 51]. Ta analogia ma pewne ograniczenia (np. druga pochodna stanu nie ma sensu), jest jednak przydatnym punktem wyjścia do myślenia o danych.

$$\text{stan}(\text{teraz}) = \int_{t=0}^{\text{teraz}} \text{strumień}(t) dt \qquad \text{strumień}(t) = \frac{d \text{stan}(t)}{dt}$$

Rysunek 11.6. Zależność między aktualnym stanem aplikacji a strumieniem zdarzeń

Jeśli trwale zapiszesz dziennik zmian, będziesz mógł odtworzyć stan. Jeżeli uznasz dziennik zdarzeń za system zapisu, a modyfikowalny stan za dane pochodne, łatwiej Ci będzie analizować przepływ danych w systemie. Pat Helland ujmuje to tak [52]:

Dziennik transakcji rejestruje wszystkie zmiany wprowadzone w bazie. Szybkie dołączanie danych to jedyny sposób modyfikowania dziennika. Z tej perspektywy zawartość bazy to pamięć podręczna z najnowszymi wartościami rekordów z dziennika. Prawdą jest dziennik. Baza to pamięć podręczna z podzbiorem dziennika. Ten podzbiór z pamięci podręcznej to najnowsza wartość każdego rekordu i jej indeks z dziennika.

Kompresja dziennika (co opisano w punkcie „Kompresja dziennika”) jest jednym ze sposobów na zmniejszenie różnicy między dziennikiem a stanem bazy. Skompresowany dziennik zawiera tylko najnowszą wersję każdego rekordu i odrzuca zastąpione wersje.

## Zalety niemodyfikowalnych zdarzeń

Niemodyfikowalność w bazach danych to dawny pomysł. Na przykład księgowi od wieków wykorzystują niemodyfikowalność w rachunkowości. Gdy przeprowadza się transakcję, zostaje ona zapisana w *księdze* umożliwiającej tylko dołączanie danych. Ta księga jest jak dziennik zdarzeń opisujący przekazywane pieniądze, towary i usługi. Obrachunki (np. zysk, strata lub bilans) są uzyskiwane na podstawie transakcji z księgi w wyniku ich zsumowania [53].

Po popełnieniu błędu księgowi nie usuwają ani nie modyfikują nieprawidłowej transakcji w księdze. Zamiast tego dodają nową transakcję kompensującą błąd (np. zwracając błędnie naliczoną opłatę). Nieprawidłowa transakcja pozostaje w księdze na zawsze, ponieważ może być potrzebna w trakcie audytu. Jeśli błędne dane uzyskane na podstawie niepoprawnej księgi zostały już opublikowane, w danych z następnego okresu rozliczeniowego znajdzie się korekta. Ten proces to w księgowości coś zupełnie normalnego [54].



Choć w systemach finansowych możliwość przeprowadzenia audytu ma wyjątkowe znaczenie, jest ona przydatna także w wielu innych systemach, które nie podlegają tak ścisłym regulacjom. W punkcie „Filozofia obsługi danych wyjściowych z procesów wsadowych” opisano, że jeśli przypadkowo zainstalujesz błędny kod, który zapisuje w bazie nieprawidłowe dane, przywrócenie stanu będzie znacznie trudniejsze, jeżeli kod może destrukcyjnie nadpisywać dane. Dzięki umożliwiającemu tylko dołączanie danych dziennikowi niemodyfikowalnych zdarzeń znacznie łatwiej jest zdiagnozować, co się stało, i przywrócić stan po problemie.

Niemodyfikowalne zdarzenia zapewniają też więcej informacji niż sam aktualny stan. Na przykład w sklepie internetowym klient może dodać produkt do koszyka, a następnie go usunąć. Choć jeśli chodzi o realizację zamówienia, pierwsze zdarzenie zostaje anulowane przez drugie, to w kontekście analiz przydatna może być wiedza o tym, że klient rozważał zakup konkretnego produktu, ale z niego zrezygnował. Możliwe, że dana osoba zechce dokonać zakupu w przyszłości albo znalazła coś innego. Te informacje są zapisywane w dzienniku zdarzeń, ale zostają utracone w bazie, która usuwa produkty po ich usunięciu z dziennika [42].

### Generowanie kilku widoków na podstawie tego samego dziennika zdarzeń

Ponadto oddzielenie modyfikowalnego stanu od niemodyfikowalnego dziennika zdarzeń pozwala wygenerować kilka różnych przeznaczonych do odczytu reprezentacji tego samego dziennika. Działa to podobnie jak uruchomienie wielu konsumentów strumienia (rysunek 11.5). Na przykład analityczna baza Druid za pomocą tego podejścia pobiera dane bezpośrednio z Kafki [55]. Ponadto Pistachio (rozproszona baza z kluczami i wartościami) używa Kafki jako dziennika zatwierdzania [56], a ujścia w narzędziu Kafka Connect mogą eksportować dane z Kafki do różnych baz i indeksów [41]. W wielu innych systemach składowania i indeksowania danych, np. na serwerach wyszukiwania, też byłoby sensowne pobieranie danych wejściowych z dziennika rozproszonego (zob. punkt „Utrzymywanie synchronizacji systemów”).

Odrębny etap przekształcania dziennika zdarzeń w bazę danych pozwala łatwiej modyfikować aplikację. Jeśli chcesz wprowadzić dodatkową funkcję, która przedstawia istniejące dane w nowy sposób, możesz wykorzystać dziennik zdarzeń do zbudowania odrębnego, zoptymalizowanego pod kątem odczytu widoku dla tej nowej funkcji. Ta funkcja może działać razem z istniejącymi systemami bez konieczności ich modyfikowania. Równoległe uruchamianie starszych i nowych systemów jest często łatwiejsze niż przeprowadzanie skomplikowanej migracji schematu w istniejącym systemie. Gdy starszy system nie jest już potrzebny, można go zamknąć i odzyskać wykorzystywane przez niego zasoby [47, 57].

Przechowywanie danych jest zwykle proste, jeśli nie musisz się martwić o to, jak będą wyglądały zapytania o nie i dostęp do nich. Wiele skomplikowanych aspektów projektu schematu, indeksowania i systemów składowania danych to wynik chęci obsługi określonych wzorców zapytań i dostępu (zob. rozdział 3.). Dlatego możesz uzyskać dużo swobody dzięki rozdzieleniu formy, w jakiej dane są zapisywane, od postaci, w jakiej są wczytywane, oraz dzięki udostępnieniu różnych widoków na potrzeby odczytu. Ten model bywa czasem nazywany **CQRS** (ang. *command query responsibility segregation*) [42, 58, 59].

Tradycyjny sposób projektowania baz i schematów opiera się na błędnym założeniu, że dane muszą być zapisywane w tej samej postaci, w jakiej będą używane w zapytaniach. Spory o normalizację i denormalizację (zob. punkt „Relacje wiele do jednego i wiele do wielu”) stają się w dużym stopniu

nieistotne, jeśli możesz przekształcić dane ze zoptymalizowanego pod kątem zapisu dziennika zdarzeń w zoptymalizowany pod kątem odczytu stan aplikacji. Denormalizacja danych w widokach zoptymalizowanych na potrzeby odczytu jest zupełnie zrozumiała, ponieważ proces przekształcania zapewnia mechanizm utrzymywania spójności widoku z dziennikiem zdarzeń.

W punkcie „Opisywanie obciążenia” przedstawiono oś czasu z Twittera będącą pamięcią podręczną tweetów osób obserwowanych przez danego użytkownika (przypomina ona skrzynkę pocztową). To następny przykład stanu zoptymalizowanego pod kątem odczytu. Oś czasu jest wysoce zdenormalizowana, ponieważ tweety danej osoby są powielane na osiach czasu wszystkich jej obserwatorów. Jednak usługa odpowiedzialna za rozgałęzianie utrzymuje synchronizację tego powielanego stanu z nowymi tweetami i nowymi obserwatorami. Dzięki temu można sobie poradzić z duplikacją.

## Kontrola współbieżności

Największą wadą metod event sourcing i CDC jest to, że konsumenci dziennika zdarzeń zwykle działają asynchronicznie. Dlatego może się zdarzyć, że użytkownik zapisze dane w dzienniku, a następnie wczyta dane z widoku opartego na dzienniku i odkryje, że jego zapis nie został tam jeszcze odzwierciedlony. Ten problem i jego możliwe rozwiązania opisano wcześniej w punkcie „Odczyt własnych zapisów”.

Jednym z rozwiązań jest aktualizowanie służącego do odczytu widoku synchronicznie z dołączaniem zdarzenia do dziennika. To wymaga transakcji wiążącej zapisy w atomową jednostkę, dlatego trzeba albo przechowywać dziennik zdarzeń i służący do odczytu widok w tym samym systemie składowania danych, albo stosować transakcję rozproszoną obejmującą różne systemy. Możesz też wykorzystać podejście opisane w punkcie „Implementowanie bazy liniowej za pomocą rozgłaszania z uporządkowaniem całkowitym”.

Z kolei generowanie aktualnego stanu na podstawie dziennika zdarzeń upraszcza niektóre aspekty kontroli współbieżności. Konieczność stosowania transakcji na wielu obiektach (zob. punkt „Operacje na pojedynczych obiektach i na wielu obiektach”) w dużym stopniu wynika z tego, że działania jednego użytkownika wymagają modyfikacji danych w kilku różnych miejscach. Dzięki metodzie event sourcing można zaprojektować zdarzenie w taki sposób, by było niezależnym opisem działań użytkownika. Te działania wymagają wtedy tylko jednego zapisu w jednym miejscu — dołączenia zdarzeń do dziennika — co łatwo jest wykonać atomowo.

Jeśli dziennik zdarzeń i stan aplikacji są podzielone na partycje w ten sam sposób (np. przetwarzanie zdarzenia z partycji 3 wymaga tylko aktualizacji partycji 3 ze stanem aplikacji), wtedy prosty jednowątkowy konsument dziennika nie wymaga kontroli współbieżności zapisów. Z projektu wynika, że konsument przetwarza tylko jedno zdarzenie naraz (zob. też „Rzeczywiste sekwencyjne wykonywanie transakcji”). Dziennik eliminuje niedeterminizm wynikający z jednoczesności, ponieważ definiuje sekwencyjne uporządkowanie zdarzeń w partycji [24]. Jeśli zdarzenie dotyczy różnych partycji ze stanem, zadanie jest trudniejsze. Ten temat opisano w rozdziale 12.

## Ograniczenia niemodyfikowalności

Wiele systemów, w których nie stosuje się metody event sourcing, też jest opartych na niemodyfikowalności. Różne bazy wewnętrznie stosują niemodyfikowalne struktury danych lub dane o wielu wersjach, aby zapewnić obsługę snapshotów z określonego momentu (zob. punkt „Indeksy i izolacja

oparta na snapshotach”). Systemy kontroli wersji takie jak Git, Mercurial i Fossil też wymagają niemodyfikowalnych danych do zachowania historii wersji plików.

W jakim zakresie wykonalne jest utrzymywanie niemodyfikowalnej historii w nieskończoność? Odpowiedź zależy od tempa modyfikowania zbioru danych. Przy niektórych typach obciążenia roboczego dane głównie są dodawane, a rzadko aktualizowane lub usuwane. Łatwo jest zapewnić niemodyfikowalność takich danych. Przy innych rodzajach obciążenia roboczego częstotliwość aktualizacji i usuwania danych w stosunkowo niewielkim zbiorze danych jest wysoka. Wtedy niemodyfikowalna historia może być nieakceptowalnie długa, problemem może się stać fragmentacja, a wydajność kompresji i odzyskiwania pamięci jest bardzo ważna ze względu na stabilność operacyjną systemu [60, 61].

Dane trzeba usuwać nie tylko ze względu na wydajność. Czasem dane trzeba skasować z przyczyn administracyjnych mimo ich niemodyfikowalności. Na przykład regulacje dotyczące prywatności mogą wymagać usuwania danych osobowych użytkownika po zamknięciu przez niego konta, ustawy związane z ochroną danych mogą wymagać kasowania błędnych informacji, a przypadkowy wyciek poufnych danych może wymagać powstrzymania.

W takich scenariuszach nie wystarczy dodać nowe zdarzenie do dziennika, aby oznaczyć, że wcześniejsze dane należy uznać za usunięte. Trzeba zmodyfikować historię i „udawać”, że określone dane nigdy nie zostały zapisane. W bazie Datomic ta funkcja jest nazywana *excision* [62], a w systemie kontroli wersji Fossil występuje podobne rozwiązanie o nazwie *shunning* [63].

Rzeczywiste usuwanie danych jest zaskakująco trudne [64], ponieważ ich kopie mogą występować w wielu miejscach. Na przykład systemy składowania danych, systemy plików i pliki SSD często zapisują dane w nowej lokalizacji, zamiast nadpisywać je w dawnym miejscu [52], a kopie zapasowe są nieraz celowo niemodyfikowalne, by zapobiec przypadkowemu usunięciu lub uszkodzeniu danych. Usuwanie często polega raczej na „utrudnianiu pobierania danych” niż na „uniemożliwianiu ich pobrania”. Mimo to czasem trzeba próbować je skasować, o czym się przekonasz z punktu „Regulacje prawne i wewnętrzne”.

## Przetwarzanie strumieniowe

Do tego miejsca rozdziału opisywano źródła strumieni (zdarzenia reprezentujące działania użytkowników, czujniki, zapisy w bazie) i przekazywanie strumieni (za pomocą bezpośrednich komunikatów, brokera komunikatów i dzienników zdarzeń).

Pozostaje omówić, co można robić ze strumieniem, gdy jest już dostępny. Można go przetwarzać. Na ogólnym poziomie są trzy możliwości:

1. Możesz pobrać dane ze zdarzeń i zapisać je w bazie, pamięci podręcznej, indeksie wyszukiwania lub w podobnym systemie składowania danych, do którego inne klienty mogą kierować zapytania. Na rysunku 11.5 pokazano, że jest to dobry sposób na zapewnienie synchronizacji bazy ze zmianami wprowadzanymi w innych częściach systemu — zwłaszcza gdy konsument strumienia jest jedynym klientem zapisującym dane w bazie. Zapis danych w systemie składowania danych to w strumieniowaniu odpowiednik procesu opisanego w punkcie „Dane wyjściowe ze wsadowych przepływów pracy”.

2. Możesz przekazywać zdarzenia do użytkowników, np. przysyłając alerty za pomocą e-maili, wysyłając powiadomienia lub strumieniowo kierować zdarzenia do działającego w czasie rzeczywistym panelu kontrolnego, gdzie są wizualizowane. Wtedy to człowiek jest ostatecznym konsumentem strumienia.
3. Możesz przetwarzać strumienie wejściowe, aby wygenerować strumienie wyjściowe. Strumienie mogą być przekazywane w potoku obejmującym kilka etapów przetwarzania, a ostatecznie trafiają do wyjścia (możliwości 1. lub 2.).

W pozostałej części rozdziału opisana została możliwość 3. — przetwarzanie strumieni w celu wygenerowania strumieni pochodnych. Fragment kodu przetwarzający strumienie w ten sposób to *operator* lub *praca*. Są one ściśle powiązane z procesami Uniksa lub pracami platformy MapReduce opisanymi w rozdziale 10. Wzorec przepływu danych jest tu podobny — system przetwarzania strumieniowego pobiera strumienie wejściowe w trybie tylko do odczytu i zapisuje dane wyjściowe do innej lokalizacji w trybie dołączania danych.

W systemach przetwarzania strumieniowego wzorce podziału danych i pracy równoległej są bardzo podobne jak w platformie MapReduce i systemach przepływu danych z rozdziału 10., dlatego te tematy nie będą tu ponownie opisywane. Podstawowe operacje mapowania (np. przekształcanie i filtrowanie rekordów) też działają tak samo.

Ważną różnicą w porównaniu z pracami wsadowymi jest to, że strumień nigdy się nie kończy. Ma to wiele skutków. Zgodnie z opisem z początku rozdziału sortowanie nieograniczonych zbiorów danych nie ma sensu, dlatego nie można zastosować złączeń z sortowaniem przez złączanie (zob. punkt „Złączenia i grupowanie na etapie redukcji”). Potrzebne są też inne mechanizmy zapewniania odporności na błędy. W przypadku pracy wsadowej działającej przez kilka minut nieudane zadanie można ponownie uruchomić od początku. Jednak jeśli praca strumieniowa działa przez kilka lat, ponowne uruchomienie jej po awarii od początku może być nieakceptowalne.

## Zastosowania przetwarzania strumieniowego

Przetwarzanie strumieniowe jest od dawna wykorzystywane na potrzeby monitorowania, gdy organizacja chce otrzymywać alerty o określonych wydarzeniach. Oto przykłady:

- Systemy wykrywania fałszerstw muszą stwierdzać, że wzorce użytkownika karty kredytowej nieoczekiwanie się zmieniły, i blokować kartę, jeśli jest prawdopodobne, że została skradziona.
- Systemy tradingowe muszą analizować zmiany cen na rynkach finansowych i przeprowadzać transakcje zgodnie z określonymi regułami.
- Systemy produkcji muszą monitorować stan maszyn w fabryce i szybko wykrywać usterki.
- Systemy wojskowe i wywiadowcze muszą śledzić poczynania potencjalnego agresora i ogłaszać alarm, jeśli występują oznaki ataku.

Rozwiązania tego rodzaju wymagają zaawansowanego dopasowywania do wzorców i wykrywania korelacji. Z czasem pojawiły się też inne zastosowania przetwarzania strumieniowego. W tym punkcie znajdziesz krótkie porównanie niektórych zastosowań tego typu.

## Przetwarzanie złożonych zdarzeń

*Przetwarzanie złożonych zdarzeń* (ang. *complex event processing* — CEP) to podejście opracowane w latach 90. na potrzeby analizowania strumieni zdarzeń. Dotyczy ono głównie zastosowań wymagających wyszukiwania określonych wzorców zdarzeń [65, 66]. Podobnie jak wyrażenia regularne umożliwiają wyszukiwanie konkretnych wzorców znaków w łańcuchu, CEP pozwala określić reguły wyszukiwania wzorców zdarzeń w strumieniu.

W systemach CEP do opisywania wykrywanych wzorców zdarzeń często używa się wysokopoziomowych deklaracyjnych języków zapytań (takie jak SQL) lub graficznych interfejsów użytkownika. Zapytania są przekazywane do systemu przetwarzania, który pobiera strumienie wejściowe i wewnętrznie uruchamia maszynę stanową odpowiedzialną za dopasowywanie. Po znalezieniu dopasowania system generuje *złożone zdarzenie* (stąd nazwa) obejmujące szczegóły wykrytego wzorca [67].

W takich systemach relacje między zapytaniami i danymi są odwrócone w porównaniu z normalnymi bazami. Baza zwykle przechowuje dane trwale i traktuje zapytania jako tymczasowe. Po otrzymaniu zapytania baza wyszukuje pasujące do niego dane, a po zakończeniu przetwarzania zapytania zapomina o nim. W systemach CEP jest na odwrót: zapytania są przechowywane długoterminowo, zdarzenia ze strumienia wejściowego stale napływają, a system próbuje dopasować zapytania do wzorca zdarzeń [68].

Implementacjami CEP są np. Esper [69], IBM InfoSphere Streams [70], Apama, TIBCO StreamBase i SQLstream. W rozproszonych systemach przetwarzania strumieniowego (takich jak Samza) w SQL-u też pojawia się obsługa zapytań deklaracyjnych dotyczących strumieni [71].

## Analizy strumieni

Innym obszarem, w którym używa się przetwarzania strumieniowego, są *analizy strumieni*. Granica między CEP a analizą strumieni jest nieostra. Zgodnie z ogólną regułą w analizach mniej istotne jest wyszukiwanie konkretnych sekwencji zdarzeń, a ważniejsze są agregacje i statystyki dotyczące dużej liczby zdarzeń. Oto przykłady:

- Pomiar częstotliwości występowania zdarzeń określonego typu (jak często zdarzenie występuje w jednostce czasu).
- Obliczanie średniej kroczącej wartości dla danego okresu.
- Porównywanie aktualnych statystyk z wynikami z wcześniejszych okresów (np. w celu wykrywania trendów lub generowania alertów po wykryciu nietypowo wysokich lub niskich wartości w porównaniu z tym samym okresem z zeszłego tygodnia).

Statystyki są zwykle obliczane dla ustalonego przedziału czasu. Na przykład możesz chcieć ustalić średnią liczbę zapytań na sekundę kierowanych do usługi w ostatnich 5 minutach i czas odpowiedzi dla percentyla 99% z tego czasu. Uśrednianie w przedziale kilku minut wygładza nieistotne wahania następujące z sekundy na sekundę, a jednocześnie zapewnia aktualny obraz zmian w ruchu. Przedział czasowy uwzględniany w agregacji to *okno*. Szczegółowe omówienie okien zawiera punkt „Wnioskowanie na temat czasu”.

Systemy analizy strumieni wykorzystują czasem algorytmy probabilistyczne, np. filtry Blooma (opisane w punkcie „Optymalizacje wydajności”) do określania przynależności do zbioru, algorytm HyperLogLog [72] do szacowania liczności i różne algorytmy szacowania percentyli (zob. punkt

„Percentyle w praktyce”). Algorytmy probabilistyczne zwracają przybliżone wyniki, mają jednak tę zaletę, że w porównaniu z algorytmami dającymi precyzyjne rezultaty wymagają znacznie mniej pamięci w systemie przetwarzania strumieni. Stosowanie algorytmów przybliżonych sprawia czasem, że ludzie sądzą, iż systemy przetwarzania strumieniowego zawsze działają stratnie i nieprecyzyjnie. To jednak nieprawda. Przetwarzanie strumieniowe nie musi wymagać przybliżeń, a algorytmy probabilistyczne są tylko optymalizacją [73].

Wiele otwartych rozproszonych platform przetwarzania strumieniowego, np. Apache Storm, Spark Streaming, Flink, Concorde, Samza i Kafka Streams [74], jest zaprojektowanych z myślą o analizach. Usługi z hostingiem z tego obszaru to Google Cloud Dataflow i Azure Stream Analytics.

### Aktualizowanie widoków zmaterializowanych

Z punktu „Strumień a bazy danych” dowiedziałeś się, że strumień zmian w bazie może posłużyć do zapewniania aktualności systemów z danymi pochodnymi (takich jak pamięć podręczna, indeksy wyszukiwania i hurtownie danych) względem źródłowej bazy danych. Można to uznać za specjalny przypadek aktualizowania *widoków zmaterializowanych* (zob. punkt „Agregowanie — kostki danych i widoki zmaterializowane”). Polega to na generowaniu nowego widoku zbioru danych, co pozwala na wydajną obsługę zapytań i aktualizowanie widoku po każdej zmianie w danych [50].

Metoda event sourcing działa podobnie. Stan aplikacji jest utrzymywany na podstawie dziennika zdarzeń. Tu stan aplikacji to też pewnego rodzaju widok zmaterializowany. Inaczej niż w scenariuszach z analizą strumieni zwykle nie wystarczy uwzględnić zdarzeń z określonego okna czasowego. Zbudowanie widoku zmaterializowanego może wymagać *wszystkich* zdarzeń z arbitralnego okresu (wyjątkiem są nieaktualne zdarzenia usunięte w ramach kompresji dziennika; zob. punkt „Kompresja dziennika”). Dlatego potrzebne jest okno rozciągające się wstecz do początku czasu.

Do aktualizowania widoku zmaterializowanego teoretycznie może posłużyć dowolny system przetwarzania strumieniowego, choć konieczność przechowywania zdarzeń w nieskończoność jest sprzeczna z założeniami będącymi podstawą niektórych platform analitycznych uwzględniających głównie okna o określonej długości. Samza i Kafka Streams obsługują tego rodzaju zastosowania, wykorzystując dostępną w Kafce obsługę kompresji dzienników [75].

### Przeszukiwanie strumieni

Obok systemów CEP, które umożliwiają wyszukiwanie wzorców obejmujących wiele zdarzeń, czasem trzeba też na podstawie złożonych kryteriów wyszukiwać pojedyncze zdarzenia (np. za pomocą zapytań z wyszukiwaniem pełnotekstowym).

Na przykład usługi monitorowania mediów subskrybują kanały informacyjne z artykułami i materiałami z różnych mediów oraz szukają wiadomości ze wzmiankami na temat firm, produktów lub określonych zagadnień. W tym celu zapytania są formułowane z góry, a następnie stale dopasowywane do strumienia wiadomości. Podobne funkcje występują w niektórych witrynach. Na przykład użytkownicy witryn z ofertami sprzedaży nieruchomości mogą zażądać powiadomień o pojawieniu się na rynku nowych mieszkań pasujących do kryteriów wyszukiwania. Mechanizm *percolator* w Elasticsearch [76] to jedna z możliwości zaimplementowania tego rodzaju przeszukiwania strumieni.

Tradycyjne wyszukiwarki najpierw indeksują dokumenty, a następnie uruchamiają zapytania z użyciem indeksu. Z kolei w przeszukiwaniu strumieni przetwarzanie przebiega odwrotnie — zapytania są zapisywane, a dokumenty sprawdzane za pomocą zapytań (tak jak w systemach CEP). W najprostszym rozwiązaniu możesz sprawdzić każdy dokument za pomocą każdego zapytania, co jednak może trwać długo, jeśli liczba zapytań jest duża. Aby zoptymalizować ten proces, można poindeksować zapytania i dokumenty, zawężając w ten sposób zbiór potencjalnie pasujących zapytań [77].

## Przekazywanie komunikatów i wywołania RPC

W punkcie „Przepływ danych za pomocą przekazywania komunikatów” opisano systemy przekazywania komunikatów jako alternatywę wobec wywołań RPC. Te systemy są wtedy używane jak mechanizm umożliwiający komunikowanie się między usługami, stosowany np. w modelu opartym na aktorach. Choć takie systemy też wykorzystują komunikaty i zdarzenia, zwykle nie traktuje się ich jak systemów przetwarzania strumieniowego:

- Platformy oparte na aktorach są przede wszystkim mechanizmem zarządzania współbieżnością i rozproszonym wykonywaniem komunikujących się ze sobą modułów, natomiast przetwarzanie strumieniowe jest głównie techniką zarządzania danymi.
- Komunikacja między aktorami jest często efemeryczna i jednostkowa, natomiast dzienniki zdarzeń są trwałe i mają wielu subskrybentów.
- Aktory mogą się komunikować w dowolny sposób (w tym za pomocą cyklicznych żądań i odpowiedzi), natomiast systemy przetwarzania strumieniowego są zwykle konfigurowane jako acykliczne potoki, w których każdy strumień jest wyjściem jednej pracy generowanym na podstawie dobrze zdefiniowanego zbioru strumieni wejściowych.

Mimo to występują pewne analogie między systemami korzystającymi z wywołań RPC (i podobnymi) oraz przetwarzaniem strumieniowym. Na przykład Apache Storm udostępnia *rozproszone wywołania RPC*, które umożliwiają przekazanie zapytań od użytkowników do zbioru węzłów przetwarzających także strumienie zdarzeń. Te zapytania są następnie łączone ze zdarzeniami ze strumieni wejściowych, a wyniki można zagregować i odesłać do użytkownika [78]. Zob. też punkt „Przetwarzanie danych z wielu partycji”.

Można też przetwarzać strumienie za pomocą platform opartych na aktorach. Wiele takich platform nie gwarantuje jednak dostarczania komunikatów po wystąpieniu awarii, dlatego przetwarzanie nie zapewnia odporności na błędy, chyba że zaimplementujesz dodatkową logikę ponawiania prób.

## Wnioskowanie na temat czasu

Systemy przetwarzania strumieniowego często muszą uwzględniać czas — zwłaszcza gdy są używane na potrzeby analiz, gdzie nieraz stosuje się okna czasowe (np. „średnia z ostatnich pięciu minut”). Może się wydawać, że znaczenie wyrażenia „ostatnie pięć minut” jest jednoznaczne i jasne. Jednak, niestety, okazuje się ono zaskakująco kłopotliwe.

W procesie wsadowym zadania błyskawicznie przetwarzają duże zbiory zdarzeń historycznych. Jeśli potrzebne są analizy na podstawie czasu, proces wsadowy musi sprawdzić znacznik czasu umieszczony w każdym zdarzeniu. Korzystanie z zegara systemowego z maszyny, gdzie działa proces wsadowy, nie ma sensu, ponieważ czas wykonywania procesu nie ma nic wspólnego z czasem wystąpienia zdarzenia.

Proces wsadowy może wczytać zdarzenia historyczne z roku w kilka minut. W większości sytuacji istotny jest tu rok danych historycznych, a nie kilka minut przetwarzania. Ponadto używanie znaczników czasu ze zdarzeń pozwala na przetwarzanie deterministyczne. Ponowne uruchomienie tego samego procesu dla tych samych danych wejściowych daje wtedy ten sam wynik (zob. punkt „Odporność na błędy”).

Z kolei wiele platform przetwarzania strumieniowego korzysta przy określaniu okien czasowych z *godziny przetwarzania* wyznaczanego za pomocą lokalnego zegara systemowego z maszyny odpowiedzialnej za przetwarzanie [79]. Zaletę tego podejścia stanowi jego prostota. Jest ono akceptowalne, jeśli między wygenerowaniem zdarzenia a jego przetwarzaniem mija niewiele czasu. To podejście nie sprawdza się, jeśli występuje duże opóźnienie przetwarzania (czyli gdy przetwarzanie może nastąpić znacznie później niż moment zdarzenia).

### Czas wystąpienia zdarzenia a godzina przetwarzania

Jest wiele powodów, dla których przetwarzanie może być opóźnione: kolejki, błędy sieci (zob. punkt „Zawodne sieci”), problemy z wydajnością prowadzące do przeciążenia brokera komunikatów lub procesora, ponowny rozruch konsumenta strumienia lub ponowne przetwarzanie dawnych zdarzeń (zob. punkt „Odtwarzanie starszych komunikatów”) w trakcie przywracania stanu po awarii lub po wyeliminowaniu błędu z kodu.

Ponadto opóźnienie komunikatów może prowadzić do nieprzewidywalnego uporządkowania komunikatów. Załóżmy, że użytkownik najpierw zgłasza jedno żądanie sieciowe (obsługiwane przez serwer WWW A), a następnie drugie (obsługiwane przez serwer B). A i B generują zdarzenia opisujące obsłużone żądania, jednak zdarzenie z serwera B dociera do brokera komunikatów przed zdarzeniem z serwera A. Wtedy system przetwarzania strumieniowego najpierw widzi zdarzenia z serwera B, a dopiero później zdarzenie z serwera A, choć w rzeczywistości zdarzenia wystąpiły w odwrotnej kolejności.

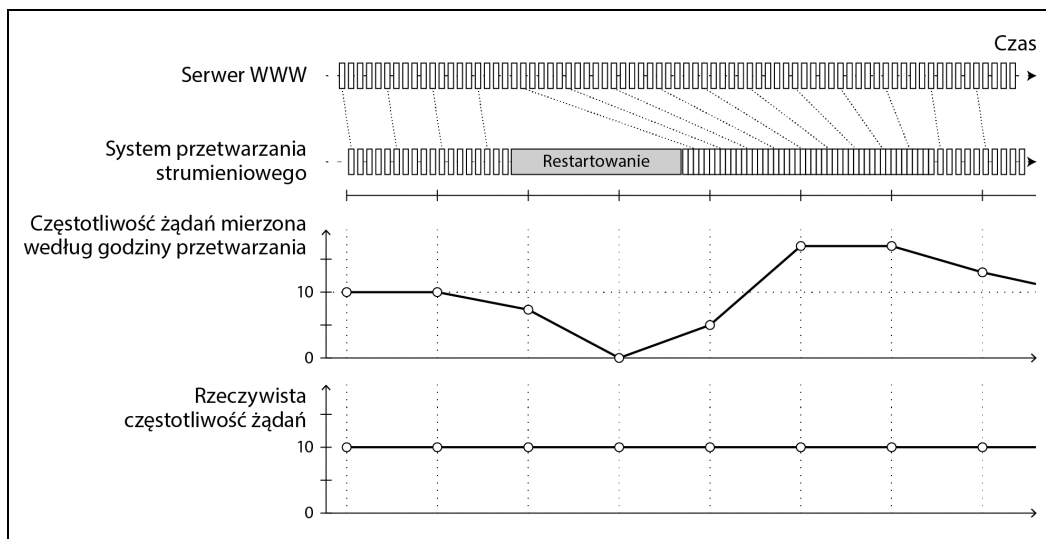
Jeśli pomocna ma być analogia, pomyśl o filmach z serii *Gwiezdne Wojny*. Część IV pojawiła się w 1977 r., część V w 1980 r., część VI w 1983 r., części I, II i III w latach 1999, 2002 i 2005, a część VII w 2015 r. [80]<sup>2</sup>. Jeżeli oglądałeś je w kolejności pojawiania się na ekranie, „przetwarzałeś” je w kolejności niezgodnej z historią. Numery części są jak znaczniki czasu zdarzeń, a data obejrzenia filmu jest godziną przetwarzania. Ludzie potrafią radzić sobie z takimi niespójnościami, jednak algorytmy przetwarzania strumieniowego trzeba pisać w specjalny sposób, aby uwzględniły problemy z czasem i uporządkowaniem.

Pomylenie czasu wystąpienia zdarzenia z godziną przetwarzania prowadzi do błędnych danych. Załóżmy, że system przetwarzania strumieniowego mierzy częstotliwość żądań (ich liczbę na sekundę). Jeśli restartujesz taki system, może on zostać zamknięty na minutę i wznowić przetwarzanie

<sup>2</sup> Za wymyślenie tej analogii dziękuję Kostasowi Kloudasowi ze społeczności skupionej wokół Flinka.



dziennika zaległych zdarzeń po ponownym rozruchu. Jeżeli mierzysz częstotliwość na podstawie godziny przetwarzania, widoczny będzie nagły anormalny skok częstotliwości z momentu przetwarzania dziennika zaległych zdarzeń, choć w rzeczywistości częstotliwość zgłaszania żądań była stała (rysunek 11.7).



Rysunek 11.7. Okna czasowe oparte na godzinie przetwarzania powodują artefakty związane ze zmianami szybkości przetwarzania

## Ustalanie zakończenia okna

Gdy okna czasowe są definiowane w kategoriach czasu wystąpienia zdarzenia, występuje kłopotliwy problem polegający na tym, że nie da się stwierdzić, czy otrzymano już wszystkie zdarzenia z określonego okna, czy jeszcze jakieś nadejdą.

Założmy, że grupujesz zdarzenia w jednominutowe okna, aby móc zliczać żądania na minutę. Uwzględniłeś już określoną liczbę zdarzeń ze znacznikami czasu z 37. minuty danej godziny, upłynął pewien czas i obecnie większość przesyłanych zdarzeń ma znaczniki z 38. i 39. minuty. Kiedy stwierdzisz, że zakończyłeś tworzenie okna dla 37. minuty i wyświetlisz dotyczący jej licznik?

Możesz stosować limit czasu i deklarować, że okno jest gotowe, jeśli przez pewien okres nie otrzymano żadnych nowych zdarzeń z tego okna. Może się jednak okazać, że niektóre zdarzenia znajdowały się w buforze innej maszyny i są opóźnione z powodu zakłóceń pracy sieci. Musisz mieć możliwość obsługi takich *opóźnionych* zdarzeń, które są przesyłane już po zadeklarowaniu zamknięcia okna. Ogólnie są dwa rozwiązania tego problemu [1]:

1. Ignorowanie opóźnionych zdarzeń, ponieważ w normalnych okolicznościach stanowią zapewne niewielki odsetek zdarzeń. Możesz śledzić wskaźnik liczby pominiętych zdarzeń i ogłaszać alert, gdy jest ich zbyt dużo.
2. Publikowanie *korekty*, czyli zaktualizowanej wartości dla okna z uwzględnieniem opóźnionych zdarzeń. Może to wymagać wycofania wcześniejszych danych wyjściowych.

W pewnych sytuacjach można zastosować specjalny komunikat oznaczający: „Od tego momentu nie pojawią się już komunikaty ze znacznikiem czasu wcześniejszym niż  $t$ ”. Konsumenci mogą wykorzystać go do zamykania okien [81]. Jeśli jednak zdarzenia są generowane przez kilku producentów z różnych maszyn i każdy z nich ustala własny minimalny znacznik czasu, konsumenci muszą śledzić poczynania każdego producenta z osobna. W tym scenariuszu dodawanie i usuwanie producentów jest trudniejsze.

### Który zegar jest używany?

Przypisywanie znaczników czasu do zdarzeń jest jeszcze trudniejsze, gdy zdarzenia mogą być buforowane w kilku miejscach systemu. Wyobraź sobie aplikację mobilną, która informuje serwer o zdarzeniach na potrzeby wskaźników użytkowania. Ta aplikacja może być wykorzystywana w czasie, gdy urządzenie jest w trybie offline. Wtedy zdarzenia są umieszczane w buforze lokalnie w urządzeniu i przesyłane na serwer, gdy połączenie internetowe stanie się dostępne (może to nastąpić godziny, a nawet dni później). Dla konsumentów tego strumienia takie zdarzenia będą się wydawały skrajnie opóźnione.

W tym kontekście znacznik czasu zdarzenia powinien odzwierciedlać czas interakcji z użytkownikiem według zegara lokalnego z urządzenia mobilnego. Jednak zegarom w urządzeniu kontrolowanym przez użytkownika często nie można ufać, ponieważ mogą zostać przypadkowo lub celowo ustawione na nieprawidłowy czas (zob. punkt „Synchronizacja i precyzja zegarów”). Czas przesłania zdarzenia na serwer (według zegara z serwera) jest prawdopodobnie bardziej precyzyjny (ponieważ serwer pozostaje pod Twoją kontrolą), ma jednak mniejszą wartość, jeśli chodzi o opis interakcji z użytkownikiem.

Jednym ze sposobów na poradzenie sobie z nieprecyzyjnymi zegarami z urządzeń jest rejestrowanie trzech znaczników czasu [82]:

- Czasu wystąpienia zdarzenia według zegara z urządzenia.
- Czasu przesłania zdarzenia na serwer według zegara z urządzenia.
- Czasu odebrania zdarzenia przez serwer według zegara z serwera.

Odjęcie drugiego znacznika czasu od trzeciego pozwala oszacować różnicę między zegarem z urządzenia a zegarem z serwera (przy założeniu, że opóźnienie sieciowe jest pomijalne w porównaniu z wymaganą precyzją znaczników czasu). Następnie możesz uwzględnić tę różnicę w znaczniku czasu zdarzenia i oszacować w ten sposób rzeczywisty czas wystąpienia zdarzenia (przyjmując, że odchylenie zegara z urządzenia nie zmieniło się między momentem wystąpienia zdarzenia a czasem przesłania go na serwer).

Ten problem dotyczy nie tylko przetwarzania strumieniowego. W przetwarzaniu wsadowym pojawiają się te same problemy z analizowaniem czasu. Po prostu są one bardziej zauważalne w kontekście strumieniowania, kiedy na upływ czasu zwraca się większą uwagę.

### Rodzaje okien

Wiesz już, jak należy rejestrować znaczniki czasu zdarzeń. Następny krok polega na ustaleniu, jak definiować okna czasowe. Okna można wykorzystać w agregacjach — np. do zliczania zdarzeń lub wyznaczania średniej wartości w ramach okna. W powszechnym użyciu są różne rodzaje okien [79, 83]:

### *Okno rozłączne (ang. tumbling window)*

Okno rozłączne ma stałą długość, a każde zdarzenie należy wtedy tylko do jednego okna. Jeśli stosujesz np. jednodominutowe okna rozłączne, wszystkie zdarzenia ze znacznikami czasu od 10:03:00 do 10:03:59 są grupowane w jedno okno, wszystkie zdarzenia od 10:04:00 do 10:04:59 tworzą następne okno itd. Możesz zaimplementować jednodominutowe okno rozłączne, rejestrując znacznik czasu każdego zdarzenia i zaokrąglając go do najbliższej minuty, aby ustalić przynależność zdarzenia do okna.

### *Okno skokowe (ang. hopping window)*

Okno skokowe też ma stałą długość, ale dopuszczalne jest zachodzenie okien na siebie w celu wygładzania danych. Na przykład okno pięciominutowe z przeskokiem jednodominutowym obejmuje zdarzenia od 10:03:00 do 10:07:59, następne okno zawiera zdarzenia od 10:04:00 do 10:08:59 itd. Okno skokowe możesz zaimplementować, obliczając najpierw jednodominutowe okna rozłączne, a następnie agregując kilka przyległych okien tego rodzaju.

### *Okno przesuwne (ang. sliding window)*

Okno przesuwne obejmuje wszystkie zdarzenia, które wystąpiły w określonej odległości między sobą. Na przykład pięciominutowe okno przesuwne może obejmować zdarzenia od 10:03:39 do 10:08:12, ponieważ są one oddalone od siebie o mniej niż 5 minut (zauważ, że w pięciominutowych oknach rozłącznych i skokowych te zdarzenia nie znajdują się w tym samym oknie, ponieważ stosowane są wtedy stałe granice). Okno przesuwne można zaimplementować za pomocą posortowanego według czasu bufora zdarzeń, z którego dawne zdarzenia są usuwane po wygaśnięciu.

### *Okno sesyjne (ang. session window)*

Okna sesyjne, w odróżnieniu od okien innego rodzaju, nie mają stałej długości. Zamiast tego są definiowane w wyniku grupowania wszystkich zdarzeń powiązanych z jednym użytkownikiem, które wystąpiły w zbliżonym czasie. Okno kończy się, gdy użytkownik jest nieaktywny przez pewien czas (np. w sytuacji, gdy przez 30 minut nie występują żadne zdarzenia). Podział na sesje jest często niezbędny w analizach witryn (zob. punkt „GROUP BY”).

## **Złączanie strumieni**

W rozdziale 10. opisano, że w pracach wsadowych zbiory danych mogą być złączane według kluczy i że takie złączenia są ważnym aspektem potoków danych. Ponieważ przetwarzanie strumieniowe to uogólnienie potoków danych na przyrostowe przetwarzanie nieograniczonych zbiorów danych, złączenia są tak samo potrzebne w strumieniach.

Jednak ponieważ w strumieniu w dowolnym momencie mogą się pojawić nowe zdarzenia, złączanie w strumieniach jest trudniejsze niż w pracach wsadowych. Aby lepiej zrozumieć sytuację, należy wprowadzić rozróżnienie na trzy typy złączeń: złączenia *strumień-strumień*, złączenia *strumień-tabela* i złączenia *tabela-tabela* [84]. W dalszych punktach każdy z tych typów jest przedstawiony za pomocą przykładu.

## **Złączenia strumień-strumień (złączanie z uwzględnieniem okna)**

Załóżmy, że udostępniasz w witrynie wyszukiwarkę i chcesz wykrywać trendy dotyczące szukanych adresów URL. Za każdym razem, gdy ktoś wprowadza zapytanie w wyszukiwarce, rejestrujesz zdarzenie obejmujące zapytanie i zwracane wyniki. Gdy użytkownik kliknie jeden z wyników, system rejestruje inne zdarzenie obejmujące to kliknięcie. Aby obliczyć współczynnik kliknięć dla każdego adresu URL z wyników wyszukiwania, trzeba połączyć zdarzenia reprezentujące wyszukiwania i kliknięcia powiązane tym samym identyfikatorem sesji. Podobnie analizy są potrzebne w systemach reklamowych [85].

Kliknięcie może nigdy nie nastąpić, jeśli użytkownik zrezygnuje z wyszukiwania. Nawet jeśli kliknięcie nastąpi, czas między wyszukiwaniem a kliknięciem może się znacznie zmieniać. W wielu sytuacjach ten czas to kilka sekund, jednak może on wynosić dni lub tygodnie (jeśli użytkownik uruchomi wyszukiwanie, zapomni o zakładce w przeglądarce, a później wróci do tej zakładki i kliknie wynik). Z powodu zmiennego opóźnienia w sieci zdarzenie kliknięcia może się pojawić przed zdarzeniem wyszukiwania. Możesz wybrać odpowiednie okno dla złączania i np. złączać kliknięcia z wyszukiwaniem, jeśli różnica czasowa między nimi wynosi najwyżej godzinę.

Zauważ, że umieszczanie informacji o wyszukiwaniu w zdarzeniu kliknięcia nie jest odpowiednikiem złączania zdarzeń. Informuje to tylko o sytuacjach, w których użytkownik kliknął wynik wyszukiwania; nie dowiesz się w ten sposób o wyszukiwaniach, po których użytkownik nie wybrał żadnego wyniku. Aby mierzyć jakość wyszukiwania, potrzebujesz precyzyjnego współczynnika kliknięć. Wymaga to zarówno zdarzeń wyszukiwania, jak i zdarzeń kliknięć.

Aby zaimplementować złączenie tego rodzaju, system przetwarzania strumieniowego musi przechowywać *stan* — np. wszystkie zdarzenia z ostatniej godziny indeksowane za pomocą identyfikatora sesji. Gdy następuje zdarzenie wyszukiwania lub zdarzenie kliknięcia, zostaje ono dołączone do odpowiedniego indeksu, a system przetwarzania strumieniowego sprawdza też drugi indeks, aby ustalić, czy pojawiło się już inne zdarzenie o tym samym identyfikatorze sesji. Po wykryciu pasującego zdarzenia generowane jest zdarzenie z informacją, że wynik wyszukiwania został kliknięty. Jeśli zdarzenie wyszukiwania wygaśnie bez pojawienia się pasującego zdarzenia kliknięcia, generowane zostanie zdarzenie z informacją, że wyniki nie zostały kliknięte.

## **Złączanie strumień-tabela (wzbogacanie strumieni)**

W punkcie „Przykład — analizy zdarzeń reprezentujących aktywność użytkownika” zobaczyłeś przykładową pracę wsadową złączającą dwa zbiory danych: zbiór zdarzeń reprezentujących aktywność użytkowników i bazę z profilami użytkowników. Naturalne jest traktowanie zdarzeń reprezentujących aktywność jako strumienia i stałe wykonywanie tego samego złączenia w systemie przetwarzania strumieniowego. Dane wejściowe to strumień zdarzeń reprezentujących aktywność, które obejmują identyfikator użytkownika. Dane wyjściowe to strumień takich zdarzeń wzbogaconych o informacje z profilu użytkownika. Ten proces można nazwać *wzbogacaniem* zdarzeń o informacje z bazy.

Aby wykonać takie złączenie, proces przetwarzający strumień musi pobierać po jednym zdarzeniu reprezentującym aktywność, wyszukiwać w bazie identyfikator użytkownika ze zdarzenia i dodawać do zdarzenia informacje o profilu. Wyszukiwanie danych w bazie można zaimplementować za pomocą zapytań do zdalnej bazy. Jednak, co opisano w punkcie „Przykład — analizy zdarzeń

reprezentujących aktywność użytkownika”, tego typu zdalne zapytania są zwykle powolne i grożą przeciążeniem bazy [75].

Inne podejście polega na wczytywaniu kopii bazy do systemu przetwarzania strumieniowego, tak aby można było wykonywać zapytania lokalnie bez przesyłania danych w sieci. Ta technika wygląda bardzo podobnie do złączania z haszowaniem opisanego w punkcie „Złączanie po stronie mappera”. Lokalna kopia bazy może się znajdować w przechowywanej w pamięci tablicy z haszowaniem, jeśli jest wystarczająco mała. Można też zastosować indeks na dysku lokalnym.

Różnica w porównaniu z pracami wsadowymi polega na tym, że w pracy wsadowej jako dane wejściowe używany jest snapshot bazy z określonego momentu. Z kolei system przetwarzania strumieniowego działa długo, a zawartość bazy zwykle w tym czasie się zmienia. Wymaga to aktualizowania lokalnej kopii bazy w tym systemie. Ten problem można rozwiązać za pomocą metody CDC. System przetwarzania strumieniowego może subskrybować dziennik zmian z bazy z profilami użytkowników, a także strumień zdarzeń reprezentujących aktywność. Gdy profil jest tworzony lub modyfikowany, system przetwarzania strumieniowego aktualizuje swoją lokalną kopię bazy. Złączane są więc dwa strumienie — zdarzeń reprezentujących aktywność i aktualizacji profili.

Złączenia strumień-tabela bardzo przypominają złączenia strumień-strumień. Największa różnica polega na tym, że w strumieniu ze zmianami z tabeli złączenia używa okna, które sięga do „początku czasu” (teoretycznie okno to jest nieskończone) i w którym nowsze wersje rekordów nadpisują starsze. Gdy dane wejściowe to strumień, złączenie może w ogóle nie używać okna.

### **Złączenia tabela-tabela (zarządzanie widokiem zmaterializowanym)**

Zastanów się nad omówionym w punkcie „Opis obciążenia” przykładem dotyczącym osi czasu z Twittera. Stwierdzono tam, że gdy użytkownik chce wyświetlić oś czasu, iteracyjne pobieranie wszystkich osób obserwowanych przez użytkownika, wyszukiwanie ich najnowszych tweetów i scalanie tych ostatnich jest zbyt kosztowne.

Zamiast tego warto stosować pamięć podręczną dla osi czasu — swoistą „skrzynkę odbiorczą” dla poszczególnych użytkowników, w której zapisywane są przeznaczone dla nich tweety. Dzięki temu odczyt osi czasu wymaga tylko jednego wyszukiwania. Materializowanie i aktualizowanie tej pamięci podręcznej wymaga przetwarzania zdarzeń w następujący sposób:

- Gdy użytkownik  $u$  zamieszcza nowy tweet, ta wiadomość jest dodawana do osi czasu każdego użytkownika obserwującego  $u$ .
- Gdy użytkownik usuwa tweet, ta wiadomość jest usuwana z osi czasu wszystkich użytkowników.
- Gdy użytkownik  $u_1$  rozpoczyna obserwowanie użytkownika  $u_2$ , nowe tweety użytkownika  $u_2$  są dodawane do osi czasu użytkownika  $u_1$ .
- Gdy użytkownik  $u_1$  przestaje obserwować użytkownika  $u_2$ , tweety użytkownika  $u_2$  są usuwane z osi czasu użytkownika  $u_1$ .

Aby zaimplementować takie zarządzanie pamięcią podręczną w systemie przetwarzania strumieniowego, potrzebne są strumienie zdarzeń związane z tweetami (ich wysyłaniem i usuwaniem) oraz relacją obserwowania (rozpoczynaniem i kończeniem obserwowania). Proces przetwarzający strumień musi przechowywać bazę obejmującą zbiór obserwatorów każdego użytkownika, aby wiedział, które osie czasu ma zaktualizować po pojawieniu się nowych tweetów [86].

Można też przyjąć, że proces przetwarzający strumień utrzymuje widok zmaterializowany na potrzeby zapytania łączącego obie tabele (tweetów i obserwacji):

```
SELECT follows.follower_id AS timeline_id,  
       array_agg(tweets.* ORDER BY tweets.timestamp DESC)  
FROM tweets  
JOIN follows ON follows.followee_id = tweets.sender_id  
GROUP BY follows.follower_id
```

Złączanie strumieni to bezpośredni odpowiednik złączania tabel w tym zapytaniu. Osie czasu są jak pamięć podręczna z wynikami dla tego zapytania aktualizowana po każdej zmianie używanej tabeli<sup>3</sup>.

## Zależności czasowe w złączeniach

Trzy opisane tu typy złączeń (strumień-strumień, strumień-tabela i tabela-tabela) mają ze sobą dużo wspólnego. Wszystkie wymagają, by system przetwarzania strumieniowego utrzymywał stan (zdarzenia wyszukiwania i zdarzenia kliknięć, profile użytkowników lub listę obserwatorów) na podstawie jednego złączanego wejścia i pobierał informacje z tego stanu na podstawie komunikatów z drugiego złączanego wejścia.

Kolejność zdarzeń związanych ze stanem ma znaczenie (ważne jest, czy najpierw zacząłeś obserwować inną osobę i później przestałeś to robić, czy odwrotnie). W dzienniku podzielonym na partycje kolejność zdarzeń w ramach jednej partycji jest zachowywana, jednak zwykle nie występują gwarancje uporządkowania dla różnych strumieni i partycji.

W tym kontekście pojawia się pytanie: jeśli zdarzenia z różnych strumieni zachodzą mniej więcej w tym samym czasie, w jakiej kolejności są przetwarzane? Jeśli w przykładowym złączaniu strumień-tabela użytkownik zaktualizuje profil, które zdarzenia reprezentujące aktywność zostaną złączone z dawnym profilem (przetworzone przed aktualizacją), a które z nowym (przetworzone po aktualizacji)? Można ująć to inaczej: jeśli stan zmienia się w czasie i łączasz dane ze stanem, z jakiego czasu stan jest używany w złączeniu [45]?

Zależności czasowe tego rodzaju mogą występować w wielu miejscach. Jeśli np. coś sprzedajesz, musisz uwzględnić na fakturach odpowiednią stawkę podatkową zależną od państwa lub stanu, rodzaju produktu i daty sprzedaży (ponieważ stawki podatkowe co pewien czas się zmieniają). Gdy łączasz transakcje z tabelą stawek podatkowych, prawdopodobnie chcesz łączyć transakcje ze stawką z momentu sprzedaży. Jeśli ponownie przetwarzasz dane historyczne, ta stawka może być inna od aktualnej.

Jeśli kolejność zdarzeń w strumieniach jest nieokreślona, złączenie staje się niedeterministyczne [87]. To oznacza, że jeśli ponownie uruchomisz tę samą pracę dla tych samych danych wejściowych, możesz uzyskać inne wyniki. Po ponownym uruchomieniu pracy zdarzenia w strumieniach wejściowych mogą być uporządkowane w inny sposób.

---

<sup>3</sup> Jeśli potraktujesz strumień jak pochodną tabeli (tak jak na rysunku 11.6) i uznasz złączenie za iloczyn dwóch tabel  $(u \cdot v)$ , uzyskasz ciekawy efekt — strumień zmian w złączeniu zmaterializowanym będzie zgodny z regułą iloczynu:  $(u \cdot v)' = u' \cdot v + u \cdot v'$ . Można to ująć inaczej: każda zmiana w tweetach jest złączana z obecnymi obserwatorami i każda zmiana w obserwatorach jest złączana z aktualnymi tweetami [49, 50].

W hurtowniach danych ten problem jest nazywany *wolno zmieniającym się wymiarem* (ang. *slowly changing dimension* — **SCD**). Często rozwiązuje się go za pomocą unikatowych identyfikatorów wersji złączanego rekordu. Na przykład po każdej zmianie stawki podatkowej przypisywany jest jej nowy identyfikator, a faktura obejmuje identyfikator stawki z dnia sprzedaży [88, 89]. Dzięki temu złączenia są deterministyczne, jednak kompresja dziennika staje się niemożliwa, ponieważ w tabeli trzeba zachować wszystkie wersje rekordów.

## Odporność na błędy

W ostatnim punkcie rozdziału omówiono, jak zapewnić odporność systemów przetwarzania strumieniowego na błędy. W rozdziale 10. zobaczyłeś, że wbudowanie takiej odporności w platformy przetwarzania wsadowego jest stosunkowo łatwe. Jeśli zadanie w pracy platformy MapReduce zawiedzie, można je ponownie uruchomić w innej maszynie, odrzucając dane wyjściowe z nieudanego zadania. To automatyczne ponawianie prób jest możliwe, ponieważ pliki wejściowe są niemodyfikowalne, każde zadanie zapisuje dane wyjściowe w odrębnym pliku w systemie HDFS, a te dane stają się widoczne dopiero po udanym wykonaniu zadania.

Odporność na błędy w podejściu wsadowym gwarantuje przede wszystkim to, że nawet po niepowodzeniu niektórych zadań dane wyjściowe pracy wsadowej będą takie same jak w sytuacji, gdy nie wystąpiły żadne problemy. Wygląda to tak, jakby każdy rekord wejściowy został przetworzony tylko raz — jakby żadne rekordy nie zostały pominięte i żadne nie zostały przetworzone dwukrotnie. Choć ponowne uruchamianie zadań oznacza, że rekordy w rzeczywistości mogą zostać przetworzone wielokrotnie, efekt widoczny w danych wyjściowych jest taki, jakby przetworzono je raz. Ta zasada to *semantyka dokładnie jednokrotnego wykonania* (ang. *exactly-once semantics*), choć bardziej precyzyjnym określeniem byłoby *praktycznie jednokrotne wykonanie* (ang. *effectively-once*) [90].

Ten sam problem odporności na błędy pojawia się w kontekście przetwarzania strumieniowego, choć trudniej sobie z nim poradzić. Oczekiwanie na zakończenie zadania przed wyświetleniem danych wyjściowych nie jest możliwe, ponieważ strumień jest nieskończony, dlatego nigdy nie da się ukończyć jego przetwarzania.

## Mikroporcje i punkty kontrolne

Jedno z rozwiązań polega na podziale strumienia na małe bloki i traktowaniu każdego bloku jak miniaturowego procesu wsadowego. To podejście to *mikroporcje* i stosuje się je w module Spark Streaming [91]. Porcja danych obejmuje zwykle ok. 1 sekundę. Jest to wynik kompromisu: mniejsze porcje powodują wyższe koszty związane z szeregowaniem i koordynacją zadań, natomiast większe porcje oznaczają dłuższe opóźnienie w udostępnianiu wyników z systemu przetwarzania strumieniowego.

Mikroporcje pośrednio tworzą okno rozłączne równe wielkości porcji (okna są tu wyznaczane na podstawie godziny przetwarzania, a nie według znaczników czasu zdarzeń). Każda praca wymagająca większego okna musi bezpośrednio przenosić stan z jednej mikroporcji do następnej.

Odmianą tego podejścia, stosowaną w platformie Apache Flink, jest okresowe generowanie bieżących punktów kontrolnych ze stanem i zapisywanie ich w pamięci trwałej [92, 93]. Jeśli w operatorze strumienia nastąpi awaria, może on wznowić pracę od najnowszego punktu kontrolnego i odrzucić

dane wyjściowe wygenerowane między ostatnim punktem kontrolnym a awarią. Generowanie punktów kontrolnych jest aktywowane przez bariery w strumieniu komunikatów. Ta technika przypomina granice mikroporcji, ale nie wymusza powstawania okien o określonej wielkości.

W ramach platformy przetwarzania strumieniowego mikroporcje i punkty kontrolne zapewniają tę samą semantykę dokładnie jednokrotnego przetwarzania co przetwarzanie wsadowe. Jednak gdy dane wyjściowe opuszczą system przetwarzania strumieniowego (np. w wyniku zapisu w bazie, przesłania komunikatów do zewnętrznego brokera komunikatów lub przesłania e-maili), platforma nie może już usunąć danych wyjściowych powiązanych ze sprawiającą problemy porcją. Wtedy ponowne uruchomienie nieudanego zadania powoduje dwukrotne wystąpienie zewnętrznych efektów ubocznych. Same mikroporcje lub punkty kontrolne nie wystarczą, aby temu zapobiec.

### Jeszcze o zatwierdzaniu atomowym

Aby zapewnić pozór dokładnie jednokrotnego przetwarzania w obliczu błędów, trzeba zagwarantować, że wszystkie dane wyjściowe i efekty uboczne przetwarzania pojawiają się *wtedy i tylko wtedy*, gdy przetwarzanie zakończyło się powodzeniem. Te efekty obejmują komunikaty przekazywane do operatorów z dalszych etapów przetwarzania lub zewnętrznych systemów obsługi komunikatów (w tym: e-maile lub przesyłane powiadomienia), zapisy w bazie, zmiany stanu operatora lub potwierdzenia komunikatów wejściowych (w tym: zmianę pozycji odnotowanej przez konsumenta w brokerze komunikatów opartym na dziennikach).

Wszystkie te rzeczy muszą albo dziać się atomowo, albo nie zachodzić w ogóle. Należy unikać utraty synchronizacji między nimi. Jeśli to podejście brzmi znajomo, jest tak, ponieważ opisano je w punkcie „Przetwarzanie komunikatów tylko raz” w kontekście transakcji rozproszonych i zatwierdzania dwuetapowego.

W rozdziale 9. opisano problemy z tradycyjnych implementacji transakcji rozproszonych (np. transakcji XA). Jednak w środowiskach z większymi ograniczeniami można wydajnie zaimplementować mechanizmy zatwierdzania atomowego. To podejście jest używane w usłudze Google Cloud Dataflow [81, 92] i bazie VoltDB [94]. Są plany, by podobne funkcje dodać do platformy Apache Kafka [95, 96]. Te implementacje (w odróżnieniu od transakcji XA) nie próbują zapewniać transakcji dla niejednorodnych technologii. Zamiast tego wewnętrznie zarządzają zarówno zmianami stanu, jak i komunikatami w ramach platformy przetwarzania strumieniowego. Koszty działania protokołu obsługi transakcji można ograniczyć, przetwarzając kilka komunikatów wejściowych w jednej transakcji.

### Idempotencja

Celem jest odrzucanie częściowych danych wyjściowych z nieudanych zadań, tak by można było bezpiecznie ponowić operację bez dwukrotnego generowania jej efektów. Jednym sposobem na osiągnięcie tego celu są transakcje rozproszone. Inny sposób to wykorzystanie *idempotencji* [97].

Operacja idempotentna to taka, którą można wykonać wielokrotnie, a efekt jest wtedy taki sam jak po jednokrotnym jej uruchomieniu. Na przykład ustawienie na jakąś wartość klucza w magazynie danych z kluczami i wartościami jest idempotentne (ponowny zapis wartości powoduje nadpisanie jej identycznymi danymi). Z kolei inkrementacja wartości licznika nie jest idempotentna (powtórzenie tej operacji oznacza, że wartość zostanie zwiększona dwukrotnie).



Nawet jeśli operacja nie jest z natury idempotentna, często za pomocą dodatkowych metadanych można jej zapewnić idempotencję. Na przykład w trakcie pobierania komunikatów z Kafki każdy komunikat ma stałą, monotonicznie rosnącą pozycję. Gdy zapisujesz wartość w zewnętrznej bazie, możesz dodać do tej wartości pozycję komunikatu, który spowodował jej ostatni zapis. Można więc stwierdzić, czy aktualizacja została już wprowadzona, i uniknąć jej ponawiania.

Obsługa stanu w rozszerzeniu Trident w systemie Storm jest oparta na podobnym pomysśle [78]. Poleganie na idempotencji wymaga przyjęcia kilku założeń: po ponownym uruchomieniu nieudane zadanie musi odtworzyć te same komunikaty w tej samej kolejności (robi to broker komunikatów oparty na dzienniku), przetwarzanie musi odbywać się deterministycznie, a żaden inny węzeł nie może jednocześnie zaktualizować danej wartości [98, 99].

W trakcie przełączania awaryjnego z jednego węzła na inny konieczne może być odgradzanie (zob. punkt „Lider i blokady”). Ma to zapobiegać zakłóceniom powodowanym przez węzeł, który uznawano za niesprawny, ale który w rzeczywistości działa. Mimo tych zastrzeżeń operacje idempotentne mogą być skutecznym sposobem na uzyskanie semantyki dokładnie jednokrotnego przetwarzania przy niewielkich kosztach.

## Odtwarzanie stanu po awarii

Wszystkie przetwarzające strumienie procesy, które wymagają stanu (np. obliczające na podstawie okien agregacje — liczby wystąpień, średnie i histogramy), muszą gwarantować możliwość odzyskania go po awarii.

Jedną z możliwości jest utrzymywanie stanu w zdalnym magazynie danych i replikacja go. Jednak kierowanie do zdalnej bazy zapytań o każdy komunikat może być niewydajne, co opisano w punkcie „Złączanie strumień-tabela (wzbogacanie strumieni)”. Inna możliwość to utrzymywanie stanu lokalnie w systemie przetwarzania strumieniowego i okresowe replikowanie go. Następnie, gdy system przetwarzania strumieniowego wznowia pracę po awarii, nowe zadanie może wczytać zreplikowany stan i wznowić przetwarzanie bez utraty danych.

Na przykład Flink okresowo rejestruje snapshoty stanu operatora i zapisuje je w pamięci trwałej (np. w systemie HDFS) [92, 93]. Samza i Kafka Streams replikują zmiany stanu, przesyłając je do dedykowanego tematu w Kafce i stosując kompresję dziennika, co przypomina metodę CDC [84, 100]. VoltDB replikuje stan, nadmiarowo przetwarzając każdy komunikat wejściowy w kilku węzłach (zob. punkt „Rzeczywiste sekwencyjne wykonywanie transakcji”).

W niektórych sytuacjach replikowanie stanu nie jest konieczne, ponieważ można go odtworzyć za pomocą strumieni wejściowych. Na przykład jeśli stan obejmuje agregacje dla krótkiego okna, odtworzenie zdarzeń wejściowych odpowiadających temu oknu może być wystarczająco szybkie. Gdy stan to lokalna replika bazy aktualizowana za pomocą metody CDC, bazę też można odtworzyć na podstawie strumienia zmian ze skompresowanego dziennika (zob. punkt „Kompresja dziennika”).

Jednak wszystkie te możliwości zależą od charakterystyki wydajności używanej infrastruktury. W niektórych systemach opóźnienie sieciowe może być niższe niż opóźnienie dostępu do dysku, a przepustowość sieci może być porównywalna z przepustowością dysku. Nie istnieje uniwersalne rozwiązanie idealne we wszystkich sytuacjach. Ponadto zalety lokalnego i zdalnego przechowywania stanu mogą się zmienić wraz z ewolucją technologii sieciowych i przechowywania danych.

# Podsumowanie

W tym rozdziale opisano strumienie zdarzeń, ich przeznaczenie i sposoby przetwarzania. Przetwarzanie strumieniowe jest pod pewnymi względami bardzo podobne do opisanego w rozdziale 10. przetwarzania wsadowego, ale wykonuje się je stale na nieograniczonych (niekończących się) strumieniach, a nie na danych wejściowych o stałej wielkości. Z tej perspektywy brokery komunikatów i dzienniki zdarzeń stanowią strumieniowy odpowiednik systemu plików.

Część rozdziału poświęcono dwóm rodzajom brokerów komunikatów:

## *Broker komunikatów w modelu AMQP i JMS*

Taki broker przypisuje poszczególne komunikaty konsumentom, a konsumenci potwierdzają pojedyncze komunikaty po ich udanym przetworzeniu. Komunikaty są usuwane z brokera po ich potwierdzeniu. To podejście jest odpowiednie dla asynchronicznych wywołań RPC (zob. też punkt „Przepływ danych za pomocą przekazywania komunikatów”) — np. w kolejkach zadań, gdzie dokładna kolejność przetwarzania komunikatów nie ma znaczenia oraz nie trzeba cofać się i ponownie wczytywać dawnych komunikatów po ich przetworzeniu.

## *Broker komunikatów oparty na dziennikach*

Taki broker przypisuje wszystkie komunikaty z partycji do tego samego węzła z konsumentem i zawsze dostarcza komunikaty w tej samej kolejności. Równoległość jest osiągana dzięki podziałowi danych na partycje. Konsumenci śledzą postęp prac, zapisując pozycję ostatniego przetworzonego komunikatu. Broker zachowuje komunikaty na dysku, dzięki czemu w razie potrzeby można się cofnąć i ponownie wczytać starsze komunikaty.

Podejście oparte na dzienniku jest podobne do dzienników replikacji używanych w bazach (zob. rozdział 5.) i w systemach składowania danych o strukturze dziennika (zob. rozdział 3.). Zobaczyłeś, że to podejście jest odpowiednie zwłaszcza w systemach przetwarzania strumieniowego, które pobierają strumień wejściowy i generują stan pochodny lub pochodne strumienie wyjściowe.

Jeśli chodzi o źródła strumieni, omówiono kilka możliwości. Naturalnie reprezentowane jako strumienie są zdarzenia reprezentujące aktywność użytkowników, czujniki generujące okresowe odczyty i kanały danych (np. z danymi rynkowymi ze świata finansów). Zobaczyłeś, że przydatne może być też traktowanie zapisów w bazie jako strumienia. Można przechwytywać dziennik zmian (czyli historię wszystkich zmian wprowadzonych w bazie) — albo bezpośrednio za pomocą metody CDC, albo pośrednio dzięki technice event sourcing. Kompresja dzienników umożliwia zachowanie pełnej kopii zawartości bazy, gdy używane są strumienie.

Reprezentowanie baz jako strumieni daje duże możliwości integracji systemów. Możesz stale utrzymywać aktualność systemów z danymi pochodnymi (np. indeksów wyszukiwania, pamięci podręcznej i systemów analitycznych), pobierając dziennik zmian i stosując go w systemie pochodnym. Możesz nawet budować widoki od nowa na podstawie istniejących danych, zaczynając od zera i przetwarzając dziennik zmian od początku do danej chwili.

Mechanizmy aktualizowania stanu za pomocą strumieni i odtwarzania komunikatów są także podstawą technik, które umożliwiają złączanie strumieni i zapewniają odporność na błędy w różnych platformach przetwarzania strumieniowego. Opisano tu kilka celów przetwarzania strumieniowego,

w tym wyszukiwanie wzorców zdarzeń (przetwarzanie zdarzeń złożonych), obliczanie agregacji dla okien (analizy strumieni) i zapewnianie aktualności systemów z danymi pochodnymi (widoki zmaterializowane).

Dalej omówiono trudności związane z analizowaniem czasu w systemach przetwarzania strumieniowego (w tym rozróżnienie na godzinę przetwarzania i znaczniki czasu wystąpienia zdarzeń). Opisano też radzenie sobie ze zdarzeniami opóźnionymi, docierającymi wtedy, gdy wydaje się, że okno jest już gotowe.

Wprowadzono rozróżnienie na trzy typy złączeń, które mogą występować w systemach przetwarzania strumieniowego:

#### *Złączenia strumień-strumień*

Oba strumienie wejściowe obejmują zdarzenia reprezentujące aktywność, a operator złączenia wyszukuje powiązane zdarzenia występujące w określonym oknie czasowym. Można np. dopasowywać dwa działania podjęte przez tego samego użytkownika w odstępie nie większym niż 30 minut. Dwoma wejściami złączenia może być ten sam strumień (jest to *samozłączenie*), jeśli chcesz znaleźć powiązane zdarzenia z tego jednego strumienia.

#### *Złączenia strumień-tabela*

Jeden strumień wejściowy obejmuje zdarzenia reprezentujące aktywność, natomiast drugim jest dziennik zmian w bazie. Dziennik zmian zapewnia aktualność lokalnej kopii bazy. Dla każdego zdarzenia reprezentującego aktywność operator złączenia kieruje zapytanie do bazy i zwraca wzbogacone zdarzenie.

#### *Złączenia tabela-tabela*

Oba strumienie wejściowe to dzienniki zmian w bazie. W tej sytuacji każda zmiana po jednej ze stron jest złączana z najnowszym stanem po drugiej stronie. Wynik to strumień zmian w zmaterializowanym widoku złączenia między dwiema tabelami.

W ostatnich punktach omówiono techniki zapewniania odporności na błędy i semantyki dokładnie jednokrotnego przetwarzania w systemie przetwarzania strumieniowego. Podobnie jak w przetwarzaniu wsadowym trzeba odrzucić częściowe dane wyjściowe z nieudanych zadań. Jednak ponieważ proces przetwarzający strumień jest długi i stale generuje dane wyjściowe, nie można w prosty sposób usuwać wszystkich danych wyjściowych. Zamiast tego można posłużyć się bardziej precyzyjnym mechanizmem przywracania stanu, wykorzystującym mikroporcje, punkty kontrolne, transakcje i zapisy idempotentne.

### **Literatura cytowana**

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers i in., *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*, „Proceedings of the VLDB Endowment”, rocznik 8, nr 12, s. 1792 – 1803, sierpień 2015 (<http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf>, <https://dl.acm.org/citation.cfm?doid=2824032.2824076>).

- [2] Harold Abelson, Gerald Jay Sussman i Julie Sussman, *Structure and Interpretation of Computer Programs*, wydanie drugie, MIT Press, 1996, ISBN: 978-0-262-51087-5 (<https://mitpress.mit.edu/sicp/>).
- [3] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui i Anne-Marie Kermarrec, *The Many Faces of Publish/Subscribe*, „ACM Computing Surveys”, rocznik 35, nr 2, s. 114 – 131, czerwiec 2003 (<http://www.cs.ru.nl/~pieter/oss/manyfaces.pdf>; <https://dl.acm.org/citation.cfm?doid=857076.857078>).
- [4] Joseph M. Hellerstein i Michael Stonebraker, *Readings in Database Systems*, wydanie czwarte, MIT Press, 2005, ISBN: 978-0-262-69314-1 (<http://redbook.cs.berkeley.edu/>).
- [5] Don Carney, Uğur Çetintemel, Mitch Cherniack i in., *Monitoring Streams — A New Class of Data Management Applications*, w: „28th International Conference on Very Large Data Bases” (VLDB), sierpień 2002 (<http://www.vldb.org/conf/2002/S07P02.pdf>).
- [6] Matthew Sackman, *Pushing Back*, *lshift.net*, 5 maja 2016 (<http://tech.labs.oliverwyman.com/blog/2016/05/05/pushing-back/>).
- [7] Vicent Martí, *Brubeck*, *a statsd-Compatible Metrics Aggregator*, *githubengineering.com*, 15 czerwca 2015 (<https://githubengineering.com/brubeck/>).
- [8] Seth Lowenberger, *MoldUDP64 Protocol Specification V 1.00*, *nasdaqtrader.com*, lipiec 2009 (<http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/moldudp64.pdf>).
- [9] Pieter Hintjens, *ZeroMQ — The Guide*, O’Reilly Media, 2013, ISBN: 978-1-449-33404-8 (<http://zguide.zeromq.org/page:all>).
- [10] Ian Malpass, *Measure Anything*, *Measure Everything*, *codeascraft.com*, 15 lutego 2011 (<https://codeascraft.com/2011/02/15/measure-anything-measure-everything/>).
- [11] Dieter Plaetinck, *25 Graphite*, *Grafana and statsd Gotchas*, *blog.raintank.io*, 3 marca 2016 (<https://grafana.com/blog/2016/03/03/25-graphite-grafana-and-statsd-gotchas/>).
- [12] Jeff Lindsay, *Web Hooks to Revolutionize the Web*, *progrium.com*, 3 maja 2007 (<http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/>).
- [13] Jim N. Gray, *Queues Are Databases*, Microsoft Research Technical Report MSR-TR-95-56, grudzień 1995 (<https://www.microsoft.com/en-us/research/publication/queues-are-databases/>).
- [14] Mark Hapner, Rich Burrridge, Rahul Sharma i in., *JSR-343 Java Message Service (JMS) 2.0 Specification*, *jms-spec.java.net*, marzec 2013 (<https://jcp.org/en/jsr/detail?id=343>).
- [15] Sanjay Aiyagari, Matthew Arrott, Mark Atwell i in., *AMQP: Advanced Message Queuing Protocol Specification*, wersja 0-9-1, listopad 2008 (<http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf>).
- [16] *Google Cloud Pub/Sub: A Google-Scale Messaging Service*, *cloud.google.com*, 2016 (<https://cloud.google.com/pubsub/architecture>).
- [17] *Apache Kafka 0.9 Documentation*, *kafka.apache.org*, listopad 2015 (<http://kafka.apache.org/documentation.html>).

- [18] Jay Kreps, Neha Narkhede i Jun Rao, *Kafka: A Distributed Messaging System for Log Processing*, w: „6th International Workshop on Networking Meets Databases” (NetDB), czerwiec 2011 (<http://notes.stephenholiday.com/Kafka.pdf>).
- [19] *Amazon Kinesis Streams Developer Guide*, [docs.aws.amazon.com](https://docs.aws.amazon.com/streams/latest/dev/introduction.html), kwiecień 2016 (<https://docs.aws.amazon.com/streams/latest/dev/introduction.html>).
- [20] Leigh Stewart i Sijie Guo, *Building DistributedLog: Twitter’s HighPerformance Replicated Log Service*, [blog.twitter.com](https://blog.twitter.com/engineering/en_us/topics/infrastructure/2015/building-distributedlog-twitter-s-high-performance-replicated-log-servic.html), 16 września 2015 ([https://blog.twitter.com/engineering/en\\_us/topics/infrastructure/2015/building-distributedlog-twitter-s-high-performance-replicated-log-servic.html](https://blog.twitter.com/engineering/en_us/topics/infrastructure/2015/building-distributedlog-twitter-s-high-performance-replicated-log-servic.html)).
- [21] *DistributedLog Documentation*, Twitter, Inc., [distributedlog.io](https://bookkeeper.apache.org/distributedlog/), maj 2016 (<https://bookkeeper.apache.org/distributedlog/>).
- [22] Jay Kreps, *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*, [engineering.linkedin.com](https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines), 27 kwietnia 2014 (<https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>).
- [23] Kartik Paramasivam, *How We’re Improving and Advancing Kafka at LinkedIn*, [engineering.linkedin.com](https://engineering.linkedin.com/apache-kafka/how-we_re-improving-and-advancing-kafka-linkedin), 2 września 2015 ([https://engineering.linkedin.com/apache-kafka/how-we\\_re-improving-and-advancing-kafka-linkedin](https://engineering.linkedin.com/apache-kafka/how-we_re-improving-and-advancing-kafka-linkedin)).
- [24] Jay Kreps, *The Log: What Every Software Engineer Should Know About RealTime Data’s Unifying Abstraction*, [engineering.linkedin.com](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying), 16 grudnia 2013 (<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>).
- [25] Shirshanka Das, Chavdar Botev, Kapil Surlaker i in., *All Aboard the Databus!*, w: „3rd ACM Symposium on Cloud Computing” (SoCC), październik 2012 (<https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/s18-das.pdf>).
- [26] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang i in., *Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services*, w: „12th USENIX Symposium on Networked Systems Design and Implementation” (NSDI), maj 2015 (<https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf>).
- [27] P.P.S. Narayan, *Sherpa Update*, [developer.yahoo.com](http://web.archive.org/web/20160801221400/https://developer.yahoo.com/blogs/ydn/sherpa-7992.html), 8 czerwca 2010 (<http://web.archive.org/web/20160801221400/https://developer.yahoo.com/blogs/ydn/sherpa-7992.html>).
- [28] Martin Kleppmann, *Bottled Water: Real-Time Integration of PostgreSQL and Kafka*, [martin.kleppmann.com](http://martin.kleppmann.com/2015/04/23/bottled-water-real-time-postgresql-kafka.html), 23 kwietnia 2015 (<http://martin.kleppmann.com/2015/04/23/bottled-water-real-time-postgresql-kafka.html>).
- [29] Ben Osheroff, *Introducing Maxwell, a mysql-to-kafka Binlog Processor*, [developer.zendesk.com](https://planet.mysql.com/entry/?id=5992024), 20 sierpnia 2015 (<https://planet.mysql.com/entry/?id=5992024>).
- [30] Randall Hauch, *Debezium 0.2.1 Released*, [debezium.io](http://debezium.io/blog/2016/06/10/Debezium-0/), 10 czerwca 2016 (<http://debezium.io/blog/2016/06/10/Debezium-0/>).

- [31] Prem Santosh Udaya Shankar, *Streaming MySQL Tables in Real-Time to Kafka*, *engineeringblog.yelp.com*, 1 sierpnia 2016 (<https://engineeringblog.yelp.com/2016/08/streaming-mysql-tables-in-real-time-to-kafka.html>).
- [32] Mongoriver, Stripe, Inc., *github.com*, wrzesień 2014 (<https://github.com/stripe/mongoriver>).
- [33] Dan Harvey, *Change Data Capture with Mongo + Kafka*, w: „Hadoop Users Group UK”, sierpień 2015 (<https://www.slideshare.net/danharvey/change-data-capture-with-mongodb-and-kafka>).
- [34] Oracle GoldenGate 12c: Real-Time Access to Real-Time Information, Oracle White Paper, marzec 2015 (<http://www.oracle.com/us/products/middleware/data-integration/oracle-goldengate-realtime-access-2031152.pdf>).
- [35] Oracle GoldenGate Fundamentals: How Oracle GoldenGate Works, Oracle Corporation, *youtube.com*, listopad 2012 (<https://www.youtube.com/watch?v=6H9NibIiPQE>).
- [36] Slava Akhmechet, *Advancing the Realtime Web*, *rethinkdb.com*, 27 stycznia 2015 (<https://rethinkdb.com/blog/realtime-web/>).
- [37] *Firebase Realtime Database Documentation*, Google, Inc., *firebase.google.com*, maj 2016 (<https://firebase.google.com/docs/database/>).
- [38] *Apache CouchDB 1.6 Documentation*, *docs.couchdb.org*, 2014 (<http://docs.couchdb.org/en/latest/>).
- [39] Matt DeBergalis, *Meteor 0.7.0: Scalable Database Queries Using MongoDB Oplog Instead of Poll-and-Diff*, *info.meteor.com*, 17 grudnia 2013 (<https://blog.meteor.com/meteor-0-7-0-scalable-database-queries-using-mongodb-oplog-instead-of-poll-and-diff-7c4166441ab8>).
- [40] *Chapter 15. Importing and Exporting Live Data*, VoltDB 6.4 User Manual, *docs.voltdb.com*, czerwiec 2016 (<https://docs.voltdb.com/UsingVoltDB/ChapExport.php>).
- [41] Neha Narkhede, *Announcing Kafka Connect: Building Large-Scale LowLatency Data Pipelines*, *confluent.io*, 18 lutego 2016 (<https://www.confluent.io/blog/announcing-kafka-connect-building-large-scale-low-latency-data-pipelines/>).
- [42] Greg Young, *CQRS and Event Sourcing*, w: „Code on the Beach”, sierpień 2014 (<https://www.youtube.com/watch?v=JHGkaShoyNs>).
- [43] Martin Fowler, *Event Sourcing*, *martinfowler.com*, 12 grudnia 2005 (<https://martinfowler.com/eaDev/EventSourcing.html>).
- [44] Vaughn Vernon, *Implementing Domain-Driven Design*, Addison-Wesley Professional, 2013, ISBN: 978-0-321-83457-7 ([https://vaughnvernon.co/?page\\_id=168](https://vaughnvernon.co/?page_id=168)).
- [45] H.V. Jagadish, Inderpal Singh Mumick i Abraham Silberschatz, *View Maintenance Issues for the Chronicle Data Model*, w: „14th ACM SIGACT-SIGMODSIGART Symposium on Principles of Database Systems” (PODS), maj 1995 (<http://www.mathcs.emory.edu/~cheung/papers/StreamDB/Histogram/1995-Jagadish-Histo.pdf>; <https://dl.acm.org/citation.cfm?doid=212433.220201>).

- [46] *Event Store 3.5.0 Documentation*, Event Store LLP, [docs.geteventstore.com](https://docs.geteventstore.com), luty 2016 (<https://eventstore.org/docs/>).
- [47] Martin Kleppmann, *Making Sense of Stream Processing*, recenzja, O'Reilly Media, maj 2016 (<http://www.oreilly.com/data/free/stream-processing.csp>).
- [48] Sander Mak, *Event-Sourced Architectures with Akka*, w: „JavaOne”, wrzesień 2014 (<https://www.slideshare.net/SanderMak/eventsourced-architectures-with-akka>).
- [49] Julian Hyde, komunikacja osobista, czerwiec 2016 (<https://twitter.com/julianhyde/status/743374145006641153>).
- [50] Ashish Gupta i Inderpal Singh Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, 1999, ISBN: 978-0-262-57122-7.
- [51] Timothy Griffin i Leonid Libkin, *Incremental Maintenance of Views with Duplicates*, w: „ACM International Conference on Management of Data” (SIGMOD), maj 1995 (<http://homepages.inf.ed.ac.uk/libkin/papers/sigmod95.pdf>; <https://dl.acm.org/citation.cfm?doid=223784.223849>).
- [52] Pat Helland, *Immutability Changes Everything*, w: „7th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2015 ([http://cidrdb.org/cidr2015/Papers/CIDR15\\_Paper16.pdf](http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf)).
- [53] Martin Kleppmann, *Accounting for Computer Scientists*, [martin.kleppmann.com](http://martin.kleppmann.com), 7 marca 2011 (<http://martin.kleppmann.com/2011/03/07/accounting-for-computer-scientists.html>).
- [54] Pat Helland, *Accountants Don't Use Erasers*, [blogs.msdn.com](http://blogs.msdn.com), 14 czerwca 2007 (<https://blogs.msdn.microsoft.com/pathelland/2007/06/14/accountants-dont-use-erasers/>).
- [55] Fangjin Yang, *Dogfooding with Druid, Samza, and Kafka: Metametrics at Metamarkets*, [metamarkets.com](http://metamarkets.com), 3 czerwca 2015 (<https://metamarkets.com/2015/dogfooding-with-druid-samza-and-kafka-metametrics-at-metamarkets/>).
- [56] Gavin Li, Jianqiu Lv i Hang Qi, *Pistachio: Co-Locate the Data and Compute for Fastest Cloud Compute*, [yahoohadoop.tumblr.com](http://yahoohadoop.tumblr.com), 13 kwietnia 2015 (<http://yahoohadoop.tumblr.com/post/116365275781/pistachio-co-locate-the-data-and-compute-for>).
- [57] Kartik Paramasivam, *Stream Processing Hard Problems — Part 1: Killing Lambda*, [engineering.linkedin.com](http://engineering.linkedin.com), 27 czerwca 2016 (<https://engineering.linkedin.com/blog/2016/06/stream-processing-hard-problems-part-1-killing-lambda>).
- [58] Martin Fowler, *CQRS*, [martinfowler.com](http://martinfowler.com), 14 lipca 2011 (<https://martinfowler.com/bliki/CQRS.html>).
- [59] Greg Young, *CQRS Documents*, [cqrs.files.wordpress.com](http://cqrs.files.wordpress.com), listopad 2010 ([https://cqrs.files.wordpress.com/2010/11/cqrs\\_documents.pdf](https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf)).
- [60] Baron Schwartz, *Immutability, MVCC, and Garbage Collection*, [xaprb.com](http://xaprb.com), 28 grudnia 2013 (<https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/>).

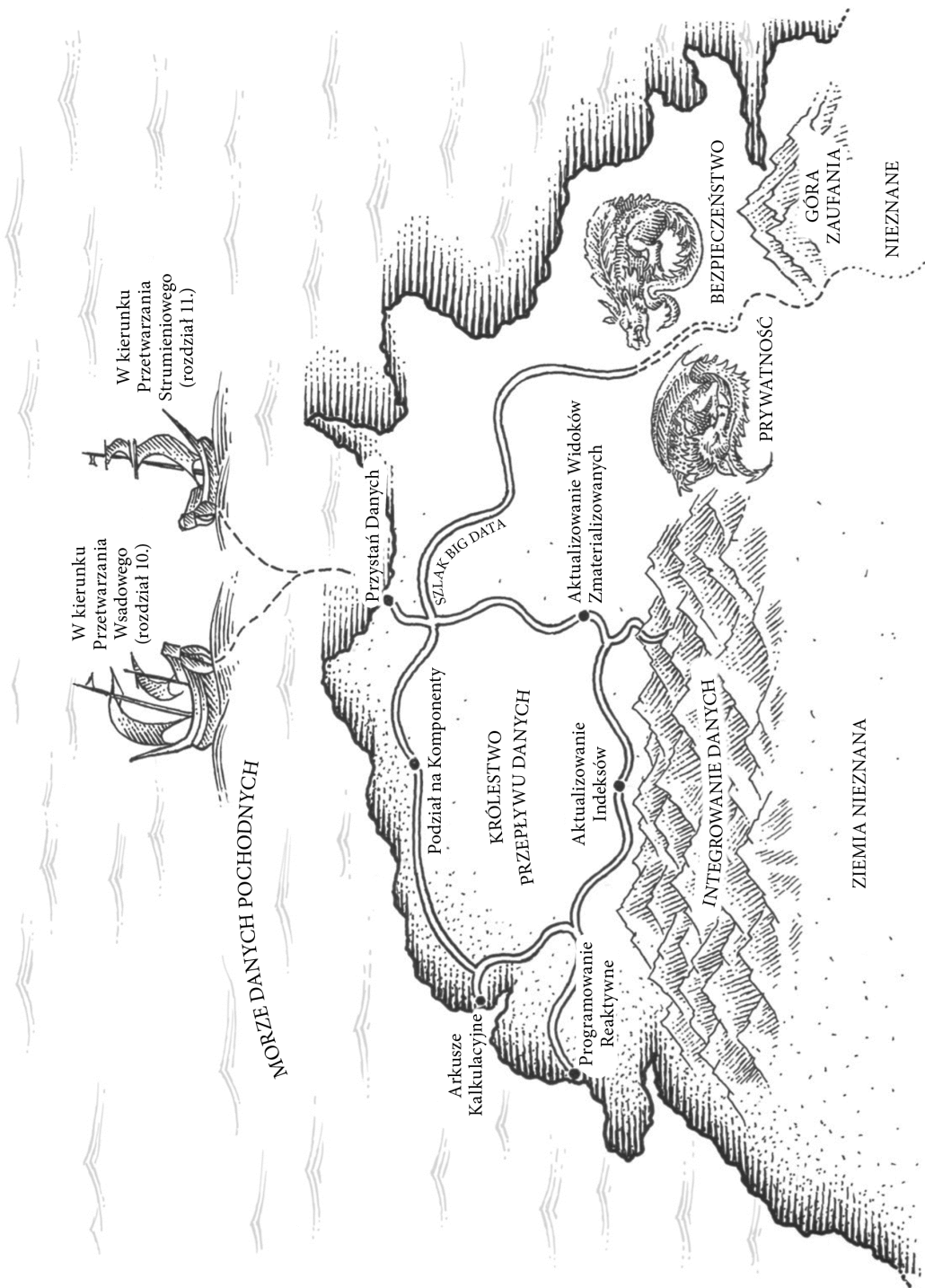
- [61] Daniel Eloff, Slava Akhmechet, Jay Kreps i in., *Re: Turning the Database Inside-out with Apache Samza*, dyskusja w serwisie Hacker News, [news.ycombinator.com](https://news.ycombinator.com/item?id=9145197), 4 marca 2015 (<https://news.ycombinator.com/item?id=9145197>).
- [62] *Datomic Development Resources: Excision*, Cognitect, Inc., [docs.datomic.com](http://docs.datomic.com/excision.html) (<http://docs.datomic.com/excision.html>).
- [63] *Fossil Documentation: Deleting Content from Fossil*, [fossil-scm.org](http://fossil-scm.org/index.html/doc/trunk/www/shunning.wiki), 2016 (<http://fossil-scm.org/index.html/doc/trunk/www/shunning.wiki>).
- [64] Jay Kreps, *The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard*, [twitter.com](https://twitter.com/jaykreps/status/582580836425330688), 30 marca 2015 (<https://twitter.com/jaykreps/status/582580836425330688>).
- [65] David C. Luckham, *What's the Difference Between ESP and CEP?*, [complexevents.com](http://www.complexevents.com), 1 sierpnia 2006 (<http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/>).
- [66] Srinath Perera, *How Is Stream Processing and Complex Event Processing (CEP) Different?*, [quora.com](https://www.quora.com/How-is-stream-processing-and-complex-event-processing-CEP-different), 3 grudnia 2015 (<https://www.quora.com/How-is-stream-processing-and-complex-event-processing-CEP-different>).
- [67] Arvind Arasu, Shivnath Babu i Jennifer Widom, *The CQL Continuous Query Language: Semantic Foundations and Query Execution*, „The VLDB Journal”, rocznik 15, nr 2, s. 121 – 142, czerwiec 2006 (<https://www.microsoft.com/en-us/research/publication/the-cql-continuous-query-language-semantic-foundations-and-query-execution/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F77607%2Ffcql.pdf>; <https://link.springer.com/article/10.1007%2Fs00778-004-0147-z>).
- [68] Julian Hyde, *Data in Flight: How Streaming SQL Technology Can Help Solve the Web 2.0 Data Crunch*, „ACM Queue”, rocznik 7, nr 11, grudzień 2009 (<http://queue.acm.org/detail.cfm?id=1667562>).
- [69] *Esper Reference, Version 5.4.0*, EsperTech, Inc., [espertech.com](http://esper.espertech.com/release-5.4.0/esper-reference/pdf/esper_reference.pdf), kwiecień 2016 ([http://esper.espertech.com/release-5.4.0/esper-reference/pdf/esper\\_reference.pdf](http://esper.espertech.com/release-5.4.0/esper-reference/pdf/esper_reference.pdf)).
- [70] Zubair Nabi, Eric Bouillet, Andrew Bainbridge i Chris Thomas, *Of Streams and Storms*, raport techniczny IBM-u, [developer.ibm.com](https://developer.ibm.com/streamsdev/wp-content/uploads/sites/15/2014/04/Streams-and-Storm-April-2014-Final.pdf), kwiecień 2014 (<https://developer.ibm.com/streamsdev/wp-content/uploads/sites/15/2014/04/Streams-and-Storm-April-2014-Final.pdf>).
- [71] Milinda Pathirage, Julian Hyde, Yi Pan i Beth Plale, *SamzaSQL: Scalable Fast Data Management with Streaming SQL*, w: „IEEE International Workshop on High-Performance Big Data Computing” (HPBDC), maj 2016 (<https://github.com/milinda/samzasql-hpbdc2016/blob/master/samzasql-hpbdc2016.pdf>; <http://ieeexplore.ieee.org/document/7530060/>).
- [72] Philippe Flajolet, Éric Fusy, Olivier Gandouet i Frédéric Meunier, *HyperLog Log: The Analysis of a Near-Optimal Cardinality Estimation Algorithm*, w: „Conference on Analysis of Algorithms” (AofA), czerwiec 2007 (<http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>).
- [73] Jay Kreps, *Questioning the Lambda Architecture*, [oreilly.com](https://www.oreilly.com/ideas/questioning-the-lambda-architecture), 2 lipca 2014 (<https://www.oreilly.com/ideas/questioning-the-lambda-architecture>).



- [74] Ian Hellström, *An Overview of Apache Streaming Technologies*, [databaseline.wordpress.com](http://databaseline.wordpress.com), 12 marca 2016 (<https://databaseline.bitbucket.io/an-overview-of-apache-streaming-technologies/>).
- [75] Jay Kreps, *Why Local State Is a Fundamental Primitive in Stream Processing*, [oreilly.com](http://www.oreilly.com), 31 lipca 2014 (<https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing>).
- [76] Shay Banon, *Percolator*, [elastic.co](http://www.elastic.co), 8 lutego 2011 (<https://www.elastic.co/blog/percolator>).
- [77] Alan Woodward i Martin Kleppmann, *Real-Time Full-Text Search with Luwak and Samza*, [martin.kleppmann.com](http://martin.kleppmann.com), 13 kwietnia 2015 (<http://martin.kleppmann.com/2015/04/13/real-time-full-text-search-luwak-samza.html>).
- [78] *Apache Storm 1.0.1 Documentation*, [storm.apache.org](http://storm.apache.org), maj 2016.
- [79] Tyler Akidau, *The World Beyond Batch: Streaming 102*, [oreilly.com](http://www.oreilly.com), 20 stycznia 2016 (<https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102>).
- [80] Stephan Ewen, *Streaming Analytics with Apache Flink*, w: „Kafka Summit”, kwiecień 2016 (<https://www.confluent.io/resources/kafka-summit-2016/advanced-streaming-analytics-apache-flink-apache-kafka/>).
- [81] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu i in., *MillWheel: Fault-Tolerant Stream Processing at Internet Scale*, w: „39th International Conference on Very Large Data Bases” (VLDB), sierpień 2013 (<https://research.google.com/pubs/pub41378.html>).
- [82] Alex Dean, *Improving Snowplow’s Understanding of Time*, [snowplowanalytics.com](http://snowplowanalytics.com), 15 września 2015 (<https://snowplowanalytics.com/blog/2015/09/15/improving-snowplows-understanding-of-time/>).
- [83] *Windowing (Azure Stream Analytics)*, Microsoft Azure Reference, [msdn.microsoft.com](http://msdn.microsoft.com), kwiecień 2016 (<https://msdn.microsoft.com/en-us/library/azure/dn835019.aspx>).
- [84] *State Management*, dokumentacja systemu Apache Samza 0.10, [samza.apache.org](http://samza.apache.org), grudzień 2015 (<http://samza.apache.org/learn/documentation/0.10/container/state-management.html>).
- [85] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das i in., *Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2013 (<https://research.google.com/pubs/pub41318.html>; <https://dl.acm.org/citation.cfm?doid=2463676.2465272>).
- [86] Martin Kleppmann, *Samza Newsfeed Demo*, [github.com](https://github.com), wrzesień 2014 (<https://github.com/ept/newsfeed>).
- [87] Ben Kirwin, *Doing the Impossible: Exactly-Once Messaging Patterns in Kafka*, [ben.kirw.in](http://ben.kirw.in), 28 listopada 2014 (<http://ben.kirw.in/2014/11/28/kafka-patterns/>).
- [88] Pat Helland, *Data on the Outside Versus Data on the Inside*, w: „2nd Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2005 (<http://cidrdb.org/cidr2005/papers/P12.pdf>).
- [89] Ralph Kimball i Margy Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, wydanie trzecie, John Wiley & Sons, 2013, ISBN: 978-1-118-53080-1.

- [90] Viktor Klang, *I'm coining the phrase 'effectively-once' for message processing with at-least-once + idempotent operations*, *twitter.com*, 20 października 2016 (<https://twitter.com/viktorklang/status/789036133434978304>).
- [91] Matei Zaharia, Tathagata Das, Haoyuan Li i in., *Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters*, w: „4th USENIX Conference in Hot Topics in Cloud Computing” (HotCloud), czerwiec 2012 (<https://www.usenix.org/system/files/conference/hotcloud12/hotcloud12-final28.pdf>).
- [92] Kostas Tzoumas, Stephan Ewen i Robert Metzger, *High-Throughput, LowLatency, and Exactly-Once Stream Processing with Apache Flink*, *data-artisans.com*, 5 sierpnia 2015 (<https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink>).
- [93] Paris Carbone, Gyula Fóra, Stephan Ewen i in., *Lightweight Asynchronous Snapshots for Distributed Dataflows*, arXiv:1506.08603 [cs.DC], 29 czerwca 2015 (<https://arxiv.org/abs/1506.08603>).
- [94] Ryan Betts i John Hugg, *Fast Data: Smart and at Scale*, recenzja, O'Reilly Media, październik 2015 (<http://www.oreilly.com/data/free/fast-data-smart-and-at-scale.csp>).
- [95] Flavio Junqueira, *Making Sense of Exactly-Once Semantics*, w: „Strata+Hadoop World London”, czerwiec 2016 (<https://conferences.oreilly.com/strata/strata-eu-2016/public/schedule/detail/49690>).
- [96] Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram Subramanian i Guozhang Wang, *KIP-98 — Exactly Once Delivery and Transactional Messaging*, *cwiki.apache.org*, listopad 2016 (<https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging>).
- [97] Pat Helland, *Idempotence Is Not a Medical Condition*, „Communications of the ACM”, rocznik 55, nr 5, s. 56, maj 2012 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.401.1539&rep=rep1&type=pdf>; <https://dl.acm.org/citation.cfm?doid=2160718.2160734>).
- [98] Jay Kreps, *Re: Trying to Achieve Deterministic Behavior on Recovery/Rewind*, e-mail na liście dyskusyjnej *samza-dev*, 9 września 2014 ([http://mail-archives.apache.org/mod\\_mbox/samza-dev/201409.mbox/%3CCA0eJiJg%2Bc7Ei%3DgzCuOz30DD3G5Hm9yFY%3DUJ6SafdNUFbvRgorg%40mail.gmail.com%3E](http://mail-archives.apache.org/mod_mbox/samza-dev/201409.mbox/%3CCA0eJiJg%2Bc7Ei%3DgzCuOz30DD3G5Hm9yFY%3DUJ6SafdNUFbvRgorg%40mail.gmail.com%3E)).
- [99] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang i David B. Johnson, *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*, „ACM Computing Surveys”, rocznik 34, nr 3, s. 375 – 408, wrzesień 2002 (<http://www.cs.utexas.edu/~lorenzo/papers/SurveyFinal.pdf>; <https://dl.acm.org/citation.cfm?doid=568522.568525>).
- [100] Adam Warski, *Kafka Streams — How Does It Fit the Stream Processing Landscape?*, *softwaremill.com*, 1 czerwca 2016 (<https://softwaremill.com/kafka-streams-how-does-it-fit-stream-landscape/>).





W kierunku  
Przetwarzania  
Wsadowego  
(rozdział 10.)

W kierunku  
Przetwarzania  
Strumieniowego  
(rozdział 11.)

MORZE DANYCH POCHODNYCH

Przystać Danych

Podział na Komponenty

KRÓLESTWO  
PRZEPŁYWU DANYCH

Programowanie  
Reaktywne

Aktualizowanie  
Indeksów

Aktualizowanie Widoków  
Zmaterializowanych

SZLAK BIG DATA

INTEGROWANIE DANYCH

PRYWATNOŚĆ

BEZPIECZEŃSTWO

GÓRA  
ZAUFIANIA

NIEZNANE

ZIEMIA NIEZNANA

# Przyszłość systemów danych

*Po pierwsze, jest rzeczą niemożliwą, by ostateczny cel tej rzeczy, która służy do osiągnięcia innego celu, polegał na zachowaniu jej istnienia. Podobnie sternik nie zmierza do zachowania powierzonego sobie okrętu jako do celu ostatecznego, gdyż okręt służy do osiągnięcia innego celu, a mianowicie do żeglugi<sup>1</sup>.*

*(Często przytacza się ten fragment w następującej formie: gdyby głównym celem kapitana było bezpieczeństwo statku, nigdy nie wypłynąłby z portu).*

— Św. Tomasz z Akwinu, *Suma teologiczna* (1265 – 1274)

Do tego miejsca w książce opisywano głównie *obecny* stan rzeczy. Poniższy ostatni rozdział zawiera spojrzenie w przyszłość i omówienie tego, jak rzeczy *powinny* wyglądać. Proponuję tu pewne pomysły i podejścia, które — jak sądzę — mogą zasadniczo usprawnić sposoby projektowania i budowania aplikacji.

Opinie i spekulacje na temat przyszłości są oczywiście subiektywne. Dlatego gdy w tym rozdziale piszę o swoich osobistych poglądach, używam pierwszej osoby. Możesz się nie zgadzać z moim zdaniem i formułować własne opinie. Mam jednak nadzieję, że idee z tego rozdziału staną się przynajmniej punktem wyjścia do wartościowych dyskusji i wprowadzą jasność w kwestiach, które często są błędnie rozumiane.

Cel tej książki został przedstawiony w rozdziale 1. — jest nim omówienie tworzenia systemów i aplikacji, które są *niezawodne, skalowalne i łatwe w konserwacji*. Te tematy powtarzały się we wszystkich rozdziałach. Omówiono np. wiele algorytmów zapewniania odporności na błędy, które pomagają poprawić niezawodność, podział na partycje zwiększający skalowalność, a także ułatwiające konserwację mechanizmy ewolucji i abstrakcji. Ten rozdział to połączenie tych wszystkich idei i wykorzystanie ich do prognozowania przyszłości. Celem jest odkrycie, jak projektować aplikacje lepsze od tych, które są dostępne obecnie — niezawodnych, prawidłowych, umożliwiających modyfikowanie i ostatecznie przydatnych ludziom.

---

<sup>1</sup> Cytat w przekładzie o. Feliksa Bednarskiego — *przyp. tłum.*

# Integrowanie danych

Jednym z motywów powtarzających się w tej książce jest to, że dla każdego problemu istnieją różne rozwiązania. Każde z nich ma różne wady i zalety oraz wymaga innych kompromisów. Na przykład w omówieniu systemów składowania danych w rozdziale 3. opisano bazy o strukturze dziennika, b-drzewa i bazy kolumnowe. W opisie replikacji w rozdziale 5. przedstawiono podejście z jednym liderem, z wieloma liderami i bez lidera.

Jeśli masz problem taki jak: „Chcę zapisać jakieś dane i później je znaleźć”, nie istnieje jedno prawidłowe rozwiązanie. Dostępnych jest wiele różnych podejść, z których każde może być odpowiednie w innych okolicznościach. W implementacji zwykle trzeba wybrać jedno konkretne rozwiązanie. Wystarczająco trudne jest już zapewnienie niezawodności i wydajności jednej ścieżki w kodzie. Próba wykonywania wszystkich zadań w jednym programie niemal gwarantuje, że implementacja będzie niskiej jakości.

Dlatego to, które narzędzie będzie najbardziej odpowiednie, też zależy od sytuacji. Każde oprogramowanie, nawet tzw. bazy do „ogólnego użytku”, są zaprojektowane pod kątem konkretnego wzorca użytkowania.

W obliczu wielu dostępnych produktów pierwszym wyzwaniem jest powiązanie ich z okolicznościami, w jakich się dobrze sprawdzają. Producenci (co zrozumiałe) niechętnie podają, do jakiego rodzaju obciążenia roboczego ich oprogramowanie jest słabo dostosowane. Mam jednak nadzieję, że wcześniejsze rozdziały pozwoliły Ci poznać odpowiednie pytania, dzięki którym będziesz potrafić czytać między wierszami i lepiej zrozumieć kompromisy.

Jednak nawet jeśli świetnie rozumiesz powiązania narzędzi ze scenariuszami ich użytkowania, pojawia się kolejna trudność. W złożonych aplikacjach dane często są używane na kilka różnych sposobów. Jest mało prawdopodobne, że jeden program będzie odpowiedni do *wszystkich* różnych sytuacji, w jakich używa się danych. Dlatego z pewnością będziesz musiał łączyć kilka różnych narzędzi, aby zapewnić odpowiednie funkcje aplikacji.

## Łączenie wyspecjalizowanych narzędzi za pomocą generowania danych

Często zdarza się, że trzeba np. zintegrować bazę OLTP z indeksem wyszukiwania pełnotekstowego, aby móc obsługiwać zapytania o dowolne słowa kluczowe. Choć istnieją bazy (np. PostgreSQL) z funkcją indeksowania pełnotekstowego, co może wystarczyć w prostych zastosowaniach [1], bardziej zaawansowane mechanizmy wyszukiwania wymagają wyspecjalizowanych narzędzi pobierania informacji. Z kolei indeksy wyszukiwania nie nadają się za bardzo na trwały system zapisu, dlatego w tak wielu aplikacjach trzeba łączyć dwa różne narzędzia, aby spełnić wszystkie wymagania.

W punkcie „Utrzymywanie synchronizacji systemów” poruszono temat integrowania systemów danych. Wraz ze wzrostem liczby różnych reprezentacji danych problem integracji staje się trudniejszy. Obok baz danych i indeksu wyszukiwania możesz potrzebować: kopii danych w systemach analitycznych (hurtowniach danych lub systemach przetwarzania wsadowego albo strumieniowego), pamięci podręcznej lub zdenormalizowanych wersji obiektów uzyskanych na podstawie pierwotnych danych, przekazywania danych w systemach uczenia maszynowego, kategoryzacji, rankingu lub rekomendacji, przesyłania powiadomień na podstawie zmian w danych.

Zaskakująco często spotykam się z takimi stwierdzeniami inżynierów oprogramowania: „Zgodnie z moim doświadczeniem 99% potrzebuje tylko X” lub „...nie potrzebuje X” (dla różnych wartości X). Uważam, że takie stwierdzenia więcej mówią o doświadczeniu danej osoby niż o rzeczywistej przydatności technologii. Zakres zastosowań danych jest zawrotnie duży. To, co jedna osoba uznaje za skomplikowaną i niepotrzebną funkcję, dla kogoś innego może być nieodzowne. Potrzeba integrowania danych staje się widoczna tylko wtedy, gdy dobrze przyjrzyś się przepływowi danych w całej organizacji i przemyślisz je.

## **Analizowanie przepływów danych**

Gdy w celu uwzględnienia różnych wzorców dostępu trzeba w kilku systemach składowania przechowywać kopie tych samych danych, należy bardzo jasno określić dane wejściowe i wyjściowe. Gdzie dane są zapisywane w pierwszej kolejności i które źródła są używane do generowania poszczególnych reprezentacji? Jak umieścić dane w odpowiednich miejscach i we właściwym formacie?

Możesz np. sprawić, żeby dane najpierw były zapisywane w bazie używanej jako system zapisu, przechwytywać zmiany wprowadzane w tej bazie (zob. punkt „Przechwytywanie zmian w danych”), a następnie w tej samej kolejności uwzględniać te zmiany w indeksie wyszukiwania. Jeśli metoda CDC to jedyne narzędzie do aktualizowania indeksu, możesz mieć pewność, że indeks jest w całości generowany na podstawie systemu zapisu, a tym samym jest z nim spójny (jeśli pominąć błędy w oprogramowaniu). Zapis w bazie jest wtedy jedynym sposobem umieszczania nowych danych wejściowych w systemie.

Umożliwienie aplikacji bezpośredniego zapisu zarówno w indeksie wyszukiwania, jak i w bazie danych powoduje problem widoczny na rysunku 11.4, gdzie dwa klienty jednocześnie przesyłają sprzeczne zapisy, a dwa systemy składowania przetwarzają je w innej kolejności. Wtedy ani baza, ani indeks wyszukiwania nie odpowiadają za określanie kolejności zapisów. Dlatego mogą podjąć sprzeczne decyzje i stać się trwale niespójne.

Jeśli możliwe jest przysyłanie wszystkich danych wejściowych od użytkowników do jednego systemu, który decyduje o kolejności wszystkich zapisów, znacznie łatwiej jest generować inne reprezentacje danych, przetwarzając zapisy w tej samej kolejności. Do tego służy replikacja maszyny stanowej opisana w punkcie „Rozgłaszanie z uporządkowaniem całkowitym”. To, czy stosujesz metodę CDC, czy dziennik z techniką event sourcing, jest mniej istotne niż zasada określania uporządkowania całkowitego.

Często można sprawić, że aktualizowanie systemu z danymi pochodnymi na podstawie dziennika zdarzeń będzie deterministyczne i idempotentne (zob. punkt „Idempotencja”), co pozwoli łatwo odzyskiwać stan po awariach.

## **Dane pochodne a transakcje rozproszone**

Klasyczny sposób utrzymywania spójności różnych systemów danych obejmuje transakcje rozproszone, co opisano w punkcie „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”. Jak systemy z danymi pochodnymi wypadają w porównaniu z transakcjami rozproszonymi?

Na abstrakcyjnym poziomie oba podejścia pozwalają osiągnąć ten sam cel odmiennymi środkami. W transakcjach rozproszonych kolejność zapisów jest ustalana za pomocą blokad powodujących wzajemne wykluczanie (zob. punkt „Blokady dwuetapowe”). W metodach CDC i event sourcing

kolejność jest określana na podstawie dziennika. W transakcjach atomowych stosuje się zatwierdzanie atomowe, aby zagwarantować, że zmiany zostaną wprowadzone tylko raz. Systemy oparte na dziennikach często wykorzystują deterministyczne ponawianie prób i idempotencję.

Największa różnica polega na tym, że systemy używające transakcji zwykle zapewniają liniowość (zob. punkt „Liniowość”), z czego wynikają przydatne gwarancje takie jak odczyt własnych zapisów (zob. punkt „Odczyt własnych zapisów”). Z kolei systemy z danymi pochodnymi często są aktualizowane asynchronicznie, dlatego domyślnie nie zapewniają takich samych gwarancji dotyczących czasu.

W środowiskach, w których operatorzy są gotowi ponosić koszty obsługi transakcji rozproszonych, te transakcje są z powodzeniem stosowane. Uważam jednak, że transakcje XA mają niską odporność na błędy i wydajność (zob. punkt „Transakcje rozproszone w praktyce”), co poważnie ogranicza ich przydatność. Uważam, że możliwe jest opracowanie lepszego protokołu transakcji rozproszonych. Jednak wprowadzenie takiego protokołu na szeroką skalę i zintegrowanie go z istniejącymi narzędziami byłoby wyzwaniem. Jest mało prawdopodobne, że stanie się to szybko.

Z powodu braku powszechnej obsługi dobrego protokołu transakcji rozproszonych sądzę, że dane pochodne w połączeniu z dziennikiem są najbardziej obiecującym sposobem integrowania różnych systemów danych. Jednak gwarancje takie jak odczyt własnych danych są przydatne, dlatego moim zdaniem nie jest słuszne przekonywanie wszystkich, że „spójność ostateczna jest nieunikniona — przełknij to i naucz się sobie z nią radzić” (przynajmniej bez dobrych wskazówek, jak to zrobić).

W punkcie „Dążenie do poprawności” opisano wybrane sposoby implementowania silniejszych gwarancji w asynchronicznych systemach z danymi pochodnymi i próby znalezienia rozwiązania pośredniego między transakcjami rozproszonymi a asynchronicznymi systemami opartymi na dziennikach.

## Ograniczenia uporządkowania całkowitego

Gdy systemy są wystarczająco małe, utworzenie dziennika zdarzeń z uporządkowaniem całkowitym jest wykonalne (czego dowodzi popularność baz z replikacją z jednym liderem, które generują właśnie taki dziennik). Jednak gdy systemy są skalowane pod kątem większego i bardziej złożonego obciążenia roboczego, zaczynają się pojawiać ograniczenia:

- W większości sytuacji budowanie dziennika z uporządkowaniem całkowitym wymaga przekazywania wszystkich zdarzeń do *jednego węzła lidera*, który ustala kolejność. Jeśli częstotliwość zgłaszania zdarzeń jest tak wysoka, że nie da się ich obsłużyć w jednej maszynie, trzeba rozdzielić zdarzenia między wiele maszyn (zob. punkt „Podział dzienników na partycje”). Kolejność zdarzeń w dwóch partycjach może być wtedy zmienna.
- Jeśli serwery są umieszczone w wielu *geograficznie rozproszonych* centrach danych, np. w celu zapewnienia odporności na niedostępność całego centrum danych, zwykle w każdym centrum działa odrębny lider, ponieważ opóźnienia sieciowe powodują, że koordynacja synchroniczna między centrami danych jest niewydajna (zob. punkt „Replikacja z wieloma liderami”). W takiej sytuacji kolejność zdarzeń pochodzących z dwóch różnych centrów danych jest niezdefiniowana.



- Gdy aplikacje są instalowane jako *mikrouslugi* (zob. punkt „Przepływ danych z użyciem usług: REST i RPC”, standardowym wyborem projektowym jest instalowanie każdej usługi (wraz z trwałym stanem) jako niezależnej jednostki. Usługi nie współdzielą trwałego stanu. Gdy dwa zdarzenia pochodzą z różnych usług, kolejność tych zdarzeń jest niezdefiniowana.
- Niektóre aplikacje przechowują stan po stronie klienta, aktualizowany natychmiast po wprowadzeniu danych wejściowych przez użytkownika (bez oczekiwania na potwierdzenie z serwera) i dostępnych nawet w trybie offline (zob. punkt „Klienci z operacjami w trybie offline”). W takich aplikacjach klienci i serwery z dużym prawdopodobieństwem zobaczą zdarzenia w odwrotnej kolejności.

Formalnie określanie uporządkowania całkowitego zdarzeń to *rozgłaszanie z uporządkowaniem całkowitym*. Jest to odpowiednik osiągania konsensusu (zob. punkt „Algorytmy osiągania konsensusu i rozgłaszanie z uporządkowaniem całkowitym”). Większość algorytmów osiągania konsensusu jest projektowana z myślą o sytuacjach, w których przepustowość jednego węzła wystarcza do przetwarzania całego strumienia zdarzeń. Takie algorytmy nie udostępniają mechanizmu pozwalającego wielu węzłom uczestniczyć w porządkowaniu zdarzeń. Otwartym problemem badawczym jest zaprojektowanie algorytmu osiągania konsensusu, który dobrze się skaluje na potrzeby przepustowości wymagającej użycia wielu węzłów i działa sprawnie w rozproszonym geograficznie środowisku.

### Porządkowanie zdarzeń w celu uchwycenia przyczynowości

W sytuacjach, gdy między zdarzeniami nie występuje związek przyczynowy, brak uporządkowania całkowitego nie jest poważnym problemem, ponieważ jednoczesne zdarzenia można uporządkować w dowolny sposób. Niektóre inne sytuacje są łatwe w obsłudze. Na przykład wiele aktualizacji tego samego obiektu można uporządkować całkowicie, umieszczając wszystkie aktualizacje obiektu o określonym identyfikatorze w tej samej partycji dziennika. Jednak zdarza się, że związki przyczynowe są bardziej subtelne (zob. też punkt „Uporządkowanie i przyczynowość”).

Wyobraź sobie, że dwoje użytkowników sieci społecznościowej pozostawało w związku, ale się rozstało. Jedna z osób usunęła drugą z listy znajomych, a następnie przesłała do pozostałych znajomych wiadomość ze skargami na byłego partnera. Ta osoba chciała, aby były partner nie zobaczył obraźliwej wiadomości, wysłanej po usunięciu go z listy znajomych.

Jednak w systemie, w którym listy znajomych są przechowywane w jednym miejscu, a wiadomości w innym, wymagająca określonej kolejności zależność między zdarzeniem *usunięcia z listy znajomych* a zdarzeniem *wysyłania wiadomości* może zostać utracona. Jeśli związek przyczynowy nie zostanie uchwycony, usługa przesyłająca powiadomienia o nowych wiadomościach może przetworzyć zdarzenie *wysyłania wiadomości* przed zdarzeniem *usunięcia z listy znajomych*, przez co powiadomienie omyłkowo trafi do byłego partnera.

W tym przykładzie powiadomienia działają jak złączenie między wiadomościami a listą znajomych, przez co mogą występować opisane wcześniej (zob. punkt „Zależności czasowe w złączeniach”) problemy z czasem typowe dla złączeń. Niestety, wydaje się, że nie istnieje proste rozwiązanie tego problemu [2, 3]. Oto możliwe punkty wyjścia:

- Logiczne znaczniki czasu pozwalają zapewnić uporządkowanie całkowite bez potrzeby koordynacji (zob. punkt „Uporządkowanie według numerów porządkowych”), dlatego mogą być pomocne w sytuacjach, gdy rozgłaszanie z uporządkowaniem całkowitym jest niewykonalne. Niezbędne jest jednak, aby odbiorcy mogli obsługiwać zdarzenia dostarczane poza kolejnością i konieczne jest przekazywanie dodatkowych metadanych.
- Jeśli możesz zapisać zdarzenie w celu zarejestrowania stanu systemu, jaki użytkownik widział przed podjęciem decyzji, i przypisać temu zdarzeniu unikatowy identyfikator, to w późniejszych zdarzeniach możesz używać tego identyfikatora do zapisania związku przyczynowego [4]. Ten pomysł został ponownie opisany w punkcie „Odczyty także są zdarzeniami”.
- Algorytmy rozwiązywania konfliktów (zob. ramka „Automatyczne rozwiązywanie konfliktów”) pomagają w przetwarzaniu zdarzeń dostarczonych w nieoczekiwanej kolejności. Są one przydatne do zarządzania stanem, ale nie wystarczają, gdy operacje powodują zewnętrzne efekty uboczne (takie jak wysłanie powiadomienia do użytkownika).

Możliwe że w przyszłości pojawią się wzorce rozwoju aplikacji, które pozwolą na wydajne ujmowanie związków przyczynowych i poprawne zarządzanie stanem danych pochodnych bez konieczności przekazywania wszystkich zdarzeń do wąskiego gardła, jakim jest rozgłaszanie z uporządkowaniem całkowitym.

## Przetwarzanie wsadowe i strumieniowe

Powiedziałbym, że celem integracji danych jest zapewnianie, iż dane trafią we właściwej postaci w odpowiednie miejsca. Wymaga to pobierania danych wejściowych, przekształcania ich, złączania, filtrowania i agregowania, trenowania modeli, sprawdzania ich i ostatecznie zapisywania w odpowiednich wyjściach. Systemy przetwarzania wsadowego i strumieniowego to narzędzia pozwalające osiągnąć ten cel.

Dane wyjściowe systemów przetwarzania wsadowego i strumieniowego to pochodne zbiory danych takie jak indeksy wyszukiwania, widoki zmaterializowane, wyświetlane użytkownikom rekomendacje, zagregowane pomiary itd. (zob. punkty „Dane wyjściowe ze wsadowych przepływów pracy” i „Zastosowania przetwarzania strumieniowego”).

W rozdziałach 10. i 11. pokazano, że w przetwarzaniu wsadowym i strumieniowym obowiązuje wiele tych samych zasad. Najważniejsza fundamentalna różnica dotyczy tego, że systemy przetwarzania strumieniowego operują na nieograniczonych zbiorach danych, natomiast dane wejściowe w procesie wsadowym mają znaną, skończoną wielkość. Występuje też wiele szczególnych różnic w sposobie implementacji poszczególnych systemów, jednak rozbieżności między kategoriami zaczynają się zacierać.

Spark obsługuje przetwarzanie strumieniowe z wykorzystaniem systemu przetwarzania wsadowego, dzieląc strumień na *mikroporcje*. Z kolei Apache Flink obsługuje przetwarzanie wsadowe z użyciem systemu przetwarzania strumieniowego [5]. Teoretycznie jeden rodzaj przetwarzania można zasymulować za pomocą drugiego, choć charakterystyka wydajności poszczególnych rozwiązań jest różna — np. mikroporcje mogą skutkować niską wydajnością, gdy używane są okna skokowe lub przesuwne [6].

## Aktualizowanie stanu pochodnego

Przetwarzanie wsadowe ma charakter wyraźnie funkcyjny (nawet jeśli kod nie jest pisany za pomocą języka funkcyjnego). Zachęca do tworzenia deterministycznych czystych funkcji, których dane wyjściowe zależą wyłącznie od danych wejściowych i które nie powodują efektów ubocznych innych niż bezpośrednio zwracane dane wyjściowe. Dane wejściowe są tu traktowane jako niemodyfikowalne, a do danych wyjściowych można tylko dołączać nowe. W przetwarzaniu strumieniowym jest podobnie, jednak operatory są tu rozbudowane w celu umożliwienia uzyskania zarządzanego stanu z odpornością na błędy (zob. punkt „Odtwarzanie stanu po awarii”).

Zasada tworzenia deterministycznych funkcji z dobrze zdefiniowanymi wejściami i wyjściami nie tylko poprawia odporność na błędy (zob. punkt „Idempotencja”), ale też upraszcza analizowanie przepływu danych w organizacji [7]. Niezależnie od tego, czy danymi pochodnymi są indeks wyszukiwania, model statystyczny, czy pamięć podręczna, warto myśleć w kategoriach potoków danych generujących jedne elementy na podstawie innych, przekazujących zmiany stanu w jednym systemie za pomocą funkcyjnego kodu aplikacji i wprowadzających zmiany w systemach pochodnych.

Teoretycznie systemy z danymi pochodnymi można aktualizować synchronicznie, podobnie jak bazy relacyjne synchronicznie aktualizują indeksy pomocnicze w ramach tej samej transakcji, która zapisuje dane w indeksowanej tabeli. Jednak to asynchroniczność zapewnia niezawodność systemów opartych na dziennikach zdarzeń. Dzięki niej błąd w jednej części systemu ma lokalny zasięg. Natomiast transakcje rozproszone są anulowane, gdy jedna ze stron zawiedzie, dlatego zwykle nasilają awarie, obejmując nią pozostałe części systemu (zob. punkt „Ograniczenia transakcji rozproszonych”).

W punkcie „Partycje i indeksy pomocnicze” zobaczyłeś, że indeksy pomocnicze często wykraczają poza granice partycji. Podzielony na partycje system z indeksami pomocniczymi albo musi przysyłać zapisy do wielu partycji (gdy indeks jest dzielony według pojęć), albo do każdej z nich (przy podziale indeksu według dokumentów). Także komunikacja między partycjami jest najbardziej niezawodna i skalowalna, jeśli indeks jest aktualizowany asynchronicznie [8] (zob. też punkt „Przetwarzanie danych w wielu partycjach”).

## Ponowne przetwarzanie danych w związku z modyfikowaniem aplikacji

Na potrzeby aktualizowania danych pochodnych użyteczne jest zarówno przetwarzanie wsadowe, jak i przetwarzanie strumieniowe. Przetwarzanie strumieniowe pozwala z niewielkim opóźnieniem odzwierciedlić zmiany z danych wyjściowych w widokach pochodnych, natomiast przetwarzanie wsadowe umożliwia ponowne przetwarzanie dużych ilości zakumulowanych danych historycznych w celu uzyskania nowych widoków istniejącego zbioru danych.

Przede wszystkim ponowne przetwarzanie istniejących danych jest dobrym mechanizmem aktualizowania systemu i modyfikowania go w celu dodania obsługi nowych funkcji i zmieniających się wymagań (zob. rozdział 4.). Bez ponownego przetwarzania modyfikacje schematu są ograniczone do prostych zmian takich jak dodanie nowego pola opcjonalnego do rekordu lub dołączenie nowego typu rekordów. Dotyczy to zarówno schematów z etapu zapisu, jak i schematów z etapu odczytu (zob. punkt „Elastyczność schematu w modelu opartym na dokumentach”). Dzięki ponownemu przetwarzaniu można zmienić strukturę zbioru danych i zastosować zupełnie inny model, aby lepiej obsługiwać nowe wymagania.

## Migracja schematów na kolei

Zakrojone na dużą skalę „migracje schematów” zdarzają się także w systemach niekomputerowych. Na przykład gdy w XIX-wiecznej Anglii budowano pierwsze tory kolejowe, było kilka konkurencyjnych rozstawów (odległości między szynami). Pociągi budowane z myślą o jednym rozstawie nie mogły jeździć po torach o innym rozstawie. Ograniczało to możliwość łączenia sieci kolejowych [9].

Po ostatecznym ustaleniu w 1846 r. jednego standardowego rozstawu tory o innych rozstawach trzeba było zmodyfikować. Jak jednak zrobić to, nie zamykając linii kolejowej na miesiące lub lata? Rozwiązanie polegało na tym, by najpierw przekształcić tory na wersję o *podwójnym* (lub *mieszanym*) rozstawie, dodając trzeci tor. Tę konwersję można było przeprowadzać stopniowo, a po jej zakończeniu po torze mogły jeździć pociągi o obu rozstawach, wykorzystując dwa z trzech torów. Ostatecznie, po przekształceniu wszystkich pociągów zgodnie ze standardowym rozstawem, tor związany z niestandardowym rozstawem można było usunąć.

„Ponowne przetwarzanie” istniejących torów w ten sposób i umożliwianie współistnienia dawnych i nowych wersji pozwala stopniowo zmieniać rozstaw w ciągu lat. Jest to jednak kosztowne przedsięwzięcie, dlatego do tej pory istnieją niestandardowe rozstawy. Na przykład rozstaw w systemie BART w San Francisco Bay Area jest inny niż w większości Stanów Zjednoczonych.

Widoki pochodne umożliwiają *stopniową* ewolucję. Jeśli chcesz zmienić strukturę zbioru danych, nie musisz przeprowadzać migracji w jednym kroku. Zamiast tego możesz równoległe utrzymywać dawny schemat i nowy schemat jako dwa niezależne pochodne widoki tych samych danych. Następnie możesz niewielkiej liczbie użytkowników udostępnić nowy widok, aby przetestować jego wydajność i wykryć błędy. Większość użytkowników cały czas jest kierowana wtedy do dawnego widoku. Stopniowo możesz zwiększać odsetek użytkowników korzystających z nowego widoku, a ostatecznie zrezygnować z dawnego widoku [10].

Piękno takiej stopniowej migracji kryje się w tym, że każdy etap procesu można łatwo cofnąć, jeśli coś się nie powiedzie. Zawsze masz działający system, do którego możesz wrócić. Dzięki zmniejszeniu zagrożenia nieodwracalnymi szkodami możesz odważniej wprowadzać zmiany i tym samym szybciej usprawniać system [11].

### Architektura lambda

Skoro przetwarzanie wsadowe służy do ponownego przetwarzania danych historycznych, a przetwarzanie strumieniowe do przetwarzania nowych aktualizacji, to jak połączyć jedno z drugim? *Architektura lambda* [12] jest propozycją z tego obszaru, której poświęcono wiele uwagi.

Główną ideą w architekturze lambda jest to, że dane przychodzące należy rejestrować, dołączając niemodyfikowalne zdarzenia do nieustannie rozrastającego się zbioru danych. Podobnie działa technika event sourcing (zob. punkt „Event sourcing”). Na podstawie tych zdarzeń tworzone są widoki zoptymalizowane pod kątem odczytu. Architektura lambda zachęca do równoległego uruchamiania dwóch różnych systemów: systemy przetwarzania wsadowego (np. MapReduce w Hadoopie) i odrębnego systemu przetwarzania strumieniowego (takiego jak Storm).

W architekturze lambda system przetwarzania strumieniowego pobiera zdarzenia i szybko generuje przybliżoną aktualizację widoku. Później system przetwarzania wsadowego pobiera *ten sam* zbiór zdarzeń i generuje poprawioną wersję widoku pochodnego. Ten projekt wynika z założenia,

że przetwarzanie wsadowe jest prostsze, a tym samym mniej narażone na błędy. Jednocześnie uważa się, że systemy przetwarzania strumieniowego są bardziej zawodne i trudniej jest zapewnić ich odporność na błędy (zob. punkt „Odporność na błędy”). Ponadto system przetwarzania strumieniowego może używać szybkich algorytmów przybliżających, a system przetwarzania wsadowego posługuje się wolniejszymi precyzyjnymi algorytmami.

Architektura lambda była ważnym pomysłem, który doprowadził do ulepszenia projektów systemów danych (zwłaszcza dzięki spopularyzowaniu zasady generowania widoków na podstawie strumieni niemodyfikowalnych zdarzeń i ponawiania przetwarzania zdarzeń w razie potrzeby). Uważam jednak, że podejście to powoduje szereg praktycznych problemów:

- Konieczność konserwacji tej samej logiki w systemach przetwarzania wsadowego i strumieniowego oznacza poważny dodatkowy wysiłek. Choć biblioteki takie jak Summingbird [13] pozwalają abstrakcyjnie zapisać obliczenia, które można wykonywać zarówno wsadowo, jak i strumieniowo, pozostają operacyjne komplikacje związane z debugowaniem, dostrajaniem i konserwacją dwóch różnych systemów [14].
- Ponieważ potoki strumieniowy i wsadowy generują odrębne dane wyjściowe, potrzebne jest scalanie w celu udzielania odpowiedzi na żądania użytkowników. To scalanie nie sprawia kłopotów, jeśli obliczenia to prosta agregacja z użyciem okna rozłącznego. Zadanie staje się znacznie trudniejsze, jeżeli widok jest generowany za pomocą dużo bardziej złożonych operacji takich jak złączenia i podział na sesje lub gdy dane wyjściowe nie są szeregami czasowymi.
- Choć możliwość ponownego przetwarzania całego historycznego zbioru danych bardzo się przydaje, to częste wykonywanie tej operacji dla dużych zbiorów danych jest kosztowne. Dlatego potok wsadowy nieraz trzeba tak skonfigurować, by przetwarzał dane w porcjach (np. dane z godziny pod koniec każdego takiego okresu), a nie w całości. Rodzi to opisane w punkcie „Wnioskowanie na temat czasu” problemy takie jak obsługa opóźnionych wartości i obsługa okien, które mogą obejmować kilka porcji danych. Stopniowe wykonywanie obliczeń wsadowych zwiększa złożoność i upodabnia je do warstwy przetwarzania strumieniowego, co jest sprzeczne z celem zachowywania jak największej prostoty warstwy wsadowej.

## Ujednolicanie przetwarzania wsadowego i strumieniowego

Nowsze dokonania pozwoliły wykorzystać zalety architektury lambda bez jej wad dzięki umożliwieniu implementacji zarówno obliczeń wsadowych (ponownego przetwarzania danych historycznych), jak i strumieniowych (przetwarzania zdarzeń, gdy napływają) w tym samym systemie [15].

Ujednolicanie przetwarzania wsadowego i strumieniowego w jednym systemie wymaga opisanych poniżej coraz szerzej dostępnych funkcji:

- Możliwość odtwarzania zdarzeń historycznych za pomocą tego samego systemu przetwarzania, który obsługuje strumień ostatnich zdarzeń. Na przykład brokery komunikatów oparte na dzienniku potrafią odtwarzać komunikaty (zob. punkt „Odtwarzanie starszych komunikatów”), a niektóre systemy przetwarzania strumieniowego potrafią wczytywać dane wejściowe z rozproszonych systemów plików takich jak HDFS.
- Semantyka dokładnie jednokrotnego przetwarzania w systemach przetwarzania strumieniowego. Chodzi o zapewnienie, że dane wyjściowe nawet po wystąpieniu błędu są takie same jak wtedy,

gdy wszystko działa prawidłowo (zob. punkt „Odporność na błędy”). Podobnie jak w przetwarzaniu wsadowym wymaga to odrzucania częściowych danych wyjściowych z nieudanych zadań.

- Narzędzia do tworzenia okien na podstawie czasu wystąpienia zdarzenia, a nie według godziny przetwarzania (ponieważ w trakcie ponownej obróbki zdarzeń historycznych godzina przetwarzania jest bez znaczenia; zob. punkt „Wnioskowanie na temat czasu”). Na przykład Apache Beam udostępnia interfejs API do zapisywania obliczeń, które można później wykonywać za pomocą narzędzi Apache Flink lub Google Cloud Dataflow.

## Podział baz danych na komponenty

Na najbardziej abstrakcyjnym poziomie bazy danych Hadoop i systemy operacyjne wykonują te same funkcje: zapisują dane oraz umożliwiają przetwarzanie ich i zgłaszanie zapytań o nie [16]. Baza zapisuje dane w rekordach zgodnych z określonym modelem danych (w wierszach tabeli, w dokumentach, w wierzchołkach grafu itd.), a system plików w systemie operacyjnym zachowuje dane w plikach, jednak w swej istocie oba te narzędzia są systemami „zarządzania informacjami” [17]. W rozdziale 10. zobaczyłeś, że ekosystem Hadoopa przypomina nieco rozproszoną wersję Uniksa.

Oczywiście występuje tu wiele istotnych w praktyce różnic. Na przykład sporo systemów plików nie radzi sobie zbyt dobrze z katalogiem obejmującym 10 mln małych plików, a baza zawierająca 10 mln małych rekordów jest czymś zupełnie normalnym i zwyczajnym. Mimo to warto zastanowić się nad podobieństwami i różnicami między systemami operacyjnymi a bazami danych.

W Uniksie i bazach relacyjnych zastosowano bardzo odmienne filozofie do problemu zarządzania informacjami. W Uniksie celem jest prezentowanie programistom logicznej, ale stosunkowo niskopoziomowej abstrakcji sprzętowej. Z kolei bazy relacyjne mają zapewniać programistom aplikacji wysokopoziomową abstrakcję ukrywającą złożone aspekty struktur danych na dysku, współbieżność, przywracanie stanu po awarii itd. W Uniksie opracowano potoki i pliki będące po prostu sekwencjami bajtów, a w kontekście baz powstały SQL i transakcje.

Które podejście jest lepsze? To oczywiście zależy od tego, czego potrzebujesz. Unix jest „prostszy” w tym sensie, że stanowi stosunkowo nieskomplikowaną nakładkę na zasoby sprzętowe. Bazy relacyjne są „prostsze” w tym ujęciu, że krótkie zapytanie deklaratywne pozwala wykorzystać rozbudowaną infrastrukturę (optymalizację zapytań, indeksy, metody złączania, kontrolę współbieżności, replikację itd.), a autor zapytania nie musi rozumieć szczegółów implementacji.

„Napięcie” między tymi filozofiami trwa od dziesięcioleci (zarówno Unix, jak i model relacyjny powstały na początku lat 70.) i wciąż nie zostało wyeliminowane. Ja interpretuję ruch NoSQL jako próbę zastosowania uniksowego podejścia z jego niskopoziomowymi abstrakcjami w obszarze rozproszonych systemów składowania danych OLTP.

W tym punkcie spróbuję pogodzić obie te filozofie w nadziei, że możliwe jest połączenie najlepszych aspektów z obu światów.

## Łączenie technologii składowania danych

W tej książce omówiono różne udostępniane przez bazy danych mechanizmy i ich działanie. Oto niektóre z nich:

- Indeksy pomocnicze, umożliwiające wydajne wyszukiwanie rekordów na podstawie wartości pola (zob. punkt „Inne struktury używane dla indeksów”).
- Widoki zmaterializowane, które są swoistą wstępnie obliczaną pamięcią podręczną z wynikami zapytań (zob. punkt „Agregacja — kostki danych i widoki zmaterializowane”).
- Dzienniki replikacji, dzięki którym kopie danych w innych węzłach mogą być aktualne (zob. punkt „Implementowanie dzienników replikacji”).
- Indeksy wyszukiwania pełnotekstowego umożliwiające wyszukiwanie słów kluczowych w tekście (zob. punkt „Wyszukiwanie pełnotekstowe i indeksy rozmyte”) i wbudowane w niektóre bazy relacyjne [1].

W rozdziałach 10. i 11. pojawiły się podobne tematy. Opisano tam budowanie indeksów wyszukiwania pełnotekstowego (zob. punkt „Dane wyjściowe ze wsadowych przepływów pracy”), zarządzanie widokami zmaterializowanymi (zob. punkt „Aktualizowanie widoków zmaterializowanych”) i replikację zmian z bazy w systemach z danymi pochodnymi (zob. punkt „Przechwytywanie zmian w danych”).

Wygląda więc na to, że istnieją analogie między mechanizmami wbudowanymi w bazy i systemy z danymi pochodnymi rozwijanymi przy użyciu systemów przetwarzania wsadowego i strumieniowego.

### Tworzenie indeksu

Pomyśl o tym, co się dzieje, gdy wywołujesz polecenie `CREATE INDEX` w celu utworzenia nowego indeksu w bazie relacyjnej. Baza musi przeskanować spójny snapshot tabeli, pobrać wszystkie wartości indeksowanego pola, posortować je i zapisać indeks. Następnie konieczne jest przetworzenie dziennika zaległych zapisów wprowadzonych od momentu wykonania spójnego snapshota (przy założeniu, że tabela nie została zablokowana na czas tworzenia indeksu, dzięki czemu można było zapisywać dane). Po wykonaniu tych operacji baza musi aktualizować indeks względem danych po każdym zapisie przeprowadzonym przez transakcję w bazie.

Ten proces jest zaskakująco podobny do konfigurowania nowej repliki pełniącej funkcję obserwatora (zob. punkt „Konfigurowanie nowych obserwatorów”), a także do uruchamiania metody CDC w systemach strumieniowania (zob. punkt „Początkowy snapshot”).

Gdy uruchamiasz polecenie `CREATE INDEX`, baza ponownie przetwarza istniejący zbiór danych (co opisano w punkcie „Ponowne przetwarzanie danych w związku z modyfikowaniem aplikacji”) i generuje indeks jako nowy widok istniejących danych. Tymi istniejącymi danymi może być snapshot ze stanem zamiast dziennika wszystkich kiedykolwiek wprowadzonych zmian, jednak te dwa elementy są ze sobą ściśle powiązane (zob. punkt „Stan, strumienie i niemodyfikowalność”).

## Metabaza z wszystkimi danymi

W tym kontekście uważam, że przepływ danych w całej organizacji zaczyna przypominać jedną wielką bazę [7]. Gdy proces wsadowy, strumieniowy lub ETL przenosi dane z jednego miejsca w inne, zmieniając przy tym ich formę, działa jak podsystem bazy zachowujący aktualność indeksów lub widoków zmaterializowanych.

W tym ujęciu systemy przetwarzania wsadowego i strumieniowego są jak złożone implementacje wyzwalaczy, procedur składowanych i procedur aktualizacji widoków zmaterializowanych. Aktualizowane w ten sposób systemy z danymi pochodnymi są jak różne rodzaje indeksów. Na przykład baza relacyjna może obsługiwać indeksy w postaci b-drzew, indeksy z haszowaniem, indeksy przestrzenne (zob. punkt „Indeksy wielokolumnowe”) i indeksy innego typu. W powstającej architekturze systemów z danymi pochodnymi te mechanizmy nie są implementowane jako funkcje jednego zintegrowanego produktu bazodanowego. Zamiast tego udostępnia się je w formie rozmaitych komponentów oprogramowania działających na innych maszynach i zarządzanych przez różne zespoły.

Dokąd te osiągnięcia zaprowadzą nas w przyszłości? Jeśli wyjdziemy z założenia, że nie istnieje jeden model danych lub format ich składowania, który jest odpowiedni dla wszystkich wzorców dostępu, myślę, że są dwie drogi łączenia różnych narzędzi do przechowywania i przetwarzania danych w spójne systemy:

### *Federacyjne bazy danych — ujednolicanie odczytów*

Można zapewnić ujednolicony interfejs zapytań dla różnych używanych systemów składowania danych i metod przetwarzania. To podejście to *federacyjne bazy danych* (ang. *federated database* lub *polystore*) [18, 19]. Aplikacje wymagające specjalnego modelu danych lub interfejsu zapytań nadal mogą bezpośrednio korzystać z systemów składowania danych, a użytkownicy chcący łączyć dane z różnych miejsc mogą to łatwo zrobić za pomocą federacyjnego interfejsu.

Federacyjny interfejs zapytań jest zgodny z relacyjnym zwyczajem budowania jednego zintegrowanego systemu z wysokopoziomowym językiem zapytań i elegancką semantyką, wymagającego jednak skomplikowanej implementacji.

### *Bazy danych z podziałem na komponenty — ujednolicanie zapisów*

Choć federacje uwzględniają zapytania tylko do odczytu w kilku różnych systemach, nie zapewniają dobrego rozwiązania problemu synchronizowania zapisów w tych systemach. Stwierdzono, że w ramach jednej bazy tworzenie spójnego indeksu to funkcja wbudowana. Gdy łączonych jest kilka systemów składowania danych, też trzeba zagwarantować, że wszystkie zmiany w danych zostaną wprowadzone we wszystkich odpowiednich miejscach — nawet po awarii. Ułatwianie niezawodnego łączenia systemów składowania danych (np. za pomocą metody CDC i dzienników zdarzeń) przypomina *podział na komponenty* mechanizmu aktualizacji indeksu bazy danych w sposób pozwalający zsynchronizować zapisy między różnymi technologiami [7, 21].

Podejście z podziałem na komponenty jest zgodne z uniksową tradycją tworzenia prostych narzędzi, które dobrze wykonują jedno zadanie [22], komunikują się za pomocą jednolitego niskopoziomowego interfejsu API (potoków) i mogą być łączone za pomocą wysokopoziomowego języka (powłoki) [16].



## Zapewnianie skuteczności podziału na komponenty

Bazy federacyjne i podział na komponenty to dwie strony tego samego medalu — budowania niezawodnych, skalowalnych i łatwych w konserwacji systemów z różnych komponentów. Zapytanie z samym odczytem w bazach federacyjnych wymaga odwzorowania jednego modelu danych na inny. Konieczne jest do tego zastanowienie, ale to wykonalne. Uważam, że synchronizowanie zapisów w kilku systemach składowania danych to trudniejszy problem inżynierski, dlatego skoncentruję się właśnie na nim.

Tradycyjne podejście do synchronizowania zapisów wymaga transakcji rozproszonych dla niejednorodnych systemów składowania danych [18]. Moim zdaniem to złe rozwiązanie (zob. punkt „Dane pochodne a transakcje rozproszone”). Transakcje w jednym systemie składowania danych lub przetwarzania strumieniowego są możliwe, sądzę jednak, że gdy dane przekazuje się między różnymi technologiami, asynchroniczny dziennik zdarzeń z idempotentnymi zapisami jest dużo solidniejszym i praktyczniejszym podejściem.

Transakcje rozproszone stosuje się np. w systemach przetwarzania strumieniowego, aby uzyskać semantykę dokładnie jednokrotnego przetwarzania (zob. punkt „Jeszcze o zatwierdzaniu atomowym”). To rozwiązanie może działać całkiem dobrze. Jednak gdy transakcja dotyczy systemów pisanych przez różne grupy osób (np. gdy dane z systemu przetwarzania strumieniowego są zapisywane w rozproszonym magazynie danych z kluczami i wartościami lub w indeksie wyszukiwania), brak standardowego protokołu transakcji znacznie utrudnia integrację. Uporządkowany dziennik zdarzeń z idempotentnymi konsumentami (zob. punkt „Idempotencja”) to znacznie prostsza abstrakcja, dlatego dużo łatwiej jest zaimplementować ją dla systemów niejednorodnych [7].

Ważną zaletą integracji z wykorzystaniem dzienników jest *luźne powiązanie* między różnymi komponentami. Uwidacznia się to na dwa sposoby:

1. Na poziomie systemu asynchroniczne strumienie zdarzeń sprawiają, że system jako całość staje się odporniejszy na przestoje lub spadek wydajności poszczególnych komponentów. Jeśli konsument zacznie działać powoli lub ulegnie awarii, dziennik zdarzeń może zapisywać komunikaty w buforze (zob. punkt „Wykorzystanie przestrzeni na dysku”), co pozwala producentowi i innym konsumentom na niezakłócone kontynuowanie pracy. Niesprawny konsument po naprawieniu może nadrobić zaległości, dlatego nie straci żadnych danych, a błąd zostanie wygaszony. Z kolei synchroniczna interakcja typowa dla transakcji rozproszonych zwykle skutkuje przekształceniem lokalnych błędów w zakrojone na dużą skalę awarie (zob. punkt „Ograniczenia transakcji rozproszonych”).
2. Jeśli chodzi o ludzi, to podział systemów danych na komponenty umożliwia tworzenie, usprawnianie i aktualizowanie różnych komponentów programowych i usług przez odrębne zespoły. Specjalizacja pozwala każdemu zespołowi skupić się na skutecznym wykonywaniu jednego zadania z wykorzystaniem dobrze zdefiniowanych interfejsów z systemów innych zespołów. Dzienniki zdarzeń zapewniają interfejs, który daje wystarczające możliwości, by uwzględnić stosunkowo mocne cechy związane ze spójnością (dzięki trwałości i uporządkowaniu zdarzeń), a jednocześnie jest na tyle ogólny, by można go stosować do niemal dowolnych danych.

## Systemy z podziałem na komponenty a systemy zintegrowane

Jeśli podział na komponenty rzeczywiście stanie się rozwiązaniem przyszłości, nie zastąpi baz w ich obecnej postaci. Będą one tak samo potrzebne jak zawsze. Bazy nadal są potrzebne do przechowywania stanu w systemach przetwarzania strumieniowego i do obsługi zapytań o dane wyjściowe z systemów przetwarzania wsadowego i strumieniowego (zob. punkty „Dane wyjściowe ze wsadowych przepływów pracy” i „Przetwarzanie strumieniowe”). W odniesieniu do określonego obciążenia roboczego nadal istotne będą wyspecjalizowane systemy obsługi zapytań. Na przykład systemy obsługi zapytań w hurtowniach danych MPP są zoptymalizowane pod kątem eksploracyjnych zapytań analitycznych i bardzo dobrze radzą sobie z takim obciążeniem roboczym (zob. punkt „Porównanie Hadoopa z rozproszonymi bazami danych”).

Problemem może być złożoność uruchamiania kilku różnych komponentów infrastruktury. Z każdym komponentem związane są procesy nauki, problemy konfiguracyjne i zawiłości operacyjne, dlatego warto instalować jak najmniej elementów. Możliwe, że jeden zintegrowany produkt pozwoli (w porównaniu z systemem składającym się z kilku narzędzi połączonych za pomocą kodu aplikacji) osiągnąć wyższą i bardziej przewidywalną wydajność dla obciążenia roboczego, dla którego został zaprojektowany [23]. W przedmowie napisałem, że budowanie narzędzi z myślą o skali, jakiej nie potrzebujesz, to marnowanie wysiłku, a ponadto może Cię zmusić do używania nieelastycznego projektu. W efekcie jest to forma przedwczesnej optymalizacji.

Celem podziału na komponenty nie jest konkutowanie ze zintegrowanymi bazami w kategorii wydajności dla konkretnego obciążenia roboczego. Ten cel to możliwość połączenia kilku różnych baz, aby uzyskać wysoką wydajność dla znacznie większego przekroju obciążeń roboczych niż jest to możliwe, gdy stosowany jest jeden produkt. Chodzi o wszechstronność, a nie o specjalizację. Z tego samego wynika tworzenie różnych modeli składowania i przetwarzania danych, co opisano w punkcie „Porównanie Hadoopa z rozproszonymi bazami danych”.

Dlatego jeśli istnieje jedna technologia, która spełnia wszystkie Twoje potrzeby, zwykle lepiej jest zastosować ten produkt, zamiast próbować budować go samodzielnie z niskopoziomowych komponentów. Zalety podziału na komponenty i łączenia są odczuwalne tylko wtedy, gdy nie istnieje jeden produkt spełniający wszystkie Twoje wymagania.

### Czego brakuje?

Narzędzia do składania systemów danych są coraz lepsze. Uważam jednak, że brakuje ważnego elementu — do tej pory nie powstał działający jak podzielona na komponenty baza odpowiednik powłoki Uniksa (czyli wysokopoziomowy język do łączenia w prosty i deklaracyjny sposób systemów składowania i przetwarzania danych).

Bardzo chciałbym mieć możliwość zastosowania np. deklaracji `mysql | elasticsearch` (analogicznie do potoków Uniksa [22]), która byłaby odpowiednikiem polecenia `CREATE INDEX`. Taka deklaracja powinna pobierać wszystkie dokumenty z bazy MySQL i indeksować je w klastrze z systemem Elasticsearch. Następnie system powinien stale przechwytywać wszystkie zmiany wprowadzane w bazie i automatycznie uwzględniać je w indeksie wyszukiwania bez konieczności pisania niestandardowego kodu aplikacji. Tego rodzaju integracja powinna być możliwa dla prawie każdego rodzaju systemów składowania lub indeksowania danych.

Świetnym rozwiązaniem byłaby też możliwość łatwiejszego wstępnego generowania i aktualizowania pamięci podręcznej. Pamiętaj, że widok zmaterializowany to w istocie wstępnie generowana pamięć podręczna. Możesz więc wyobrazić sobie tworzenie pamięci podręcznej za pomocą deklaratywnego definiowania widoków zmaterializowanych dla złożonych zapytań, w tym dla zapytań rekurencyjnych dotyczących grafów (zob. punkt „Modele danych przypominające graf”) i logiki aplikacji. W tym obszarze prowadzone są ciekawe początkowe badania, np. nad techniką *differential dataflow* [24, 25]. Mam nadzieję, że pomysły z tych badań trafią do systemów produkcyjnych.

## Projektowanie aplikacji na podstawie przepływu danych

Podejście polegające na podziale baz danych na komponenty oraz łączeniu wyspecjalizowanych systemów składowania i przetwarzania danych z kodem aplikacji zaczyna być nazywane „uzewnętrznianiem bazy danych” (ang. *database inside-out*) [26]. Ta nazwa pochodzi od tytułu wystąpienia, które wygłosiłem na konferencji w 2014 r. [27]. Jednak określanie tego podejścia „nową architekturą” to przesada. Moim zdaniem jest to raczej wzorzec projektowy i punkt wyjścia do dyskusji. Nazwa po prostu ułatwia rozmowy.

Same pomysły nie są mojego autorstwa. Stanowią one połączenie idei innych osób, od których — moim zdaniem — powinniśmy się uczyć. Moje propozycje są podobne zwłaszcza do języków *przepływu danych* takich jak Oz [28] i Juttle [29], języków FRP (ang. *functional reactive programming*) takich jak Elm [30, 31] i języków *programowania logicznego* takich jak Bloom [32]. Pojęcie *podział na komponenty* (ang. *unbundling*) w tym kontekście zaproponował Jay Kreps [7].

Nawet arkusze kalkulacyjne oferują możliwości programowania przepływów danych znacznie przewyższające to, jakie ma możliwości większość głównych języków programowania [33]. W arkuszu kalkulacyjnym możesz umieścić w jednej komórce wzór (np. na sumę komórek z innej kolumny), a po zmianie danych wejściowych powiązanych ze wzorem wynik jest ponownie automatycznie obliczany. To właśnie jest potrzebne w systemie danych. Gdy rekord w bazie się zmieni, indeks powiązany z tym rekordem ma być automatycznie aktualizowany, podobnie jak wszystkie zapisane w pamięci podręcznej widoki i agregacje zależne od tego rekordu. Nie powinien się musieć przejmować technicznymi szczegółami przebiegu tej aktualizacji. Zamiast tego powinienś móc zaufać, że ta operacja działa poprawnie.

Dlatego uważam, że twórcy większości systemów danych wciąż mogą się czegoś nauczyć na podstawie funkcji, jakie VisiCalc udostępniał już w 1979 r. [34]. Różnica w porównaniu z arkuszami danych polega na tym, że dzisiejsze systemy danych muszą być odporne na błędy i skalowalne oraz trwale przechowywać dane. Potrzebna jest też możliwość integrowania rozmaitych technologii pisanych przez różne grupy osób w innym czasie oraz ponownego wykorzystania istniejących bibliotek i usług. Nie można oczekiwać, że całe oprogramowanie będzie rozwijane za pomocą jednego konkretnego języka, platformy lub narzędzia.

W tym punkcie omawiam te pomysły i analizuję sposoby budowania aplikacji z wykorzystaniem idei podziału baz danych na komponenty i przepływów danych.

## Kod aplikacji jako funkcja generująca

Gdy jeden zbiór danych jest generowany na podstawie innego, przechodzi przez pewnego rodzaju funkcję przekształcającą. Oto przykłady:

- Indeks pomocniczy jest swoistym pochodnym zbiorem danych z prostą funkcją przekształcającą. Ta funkcja dla każdego wiersza lub dokumentu z tabeli podstawowej pobiera wartości indeksowanych kolumn lub pól i sortuje je (gdy używany jest indeks w postaci b-drzewa lub pliku SSTable, które są sortowane według kluczy, co opisano w rozdziale 3.).
- Indeks wyszukiwania pełnotekstowego jest tworzony w wyniku zastosowania różnych funkcji przetwarzania języka naturalnego takich jak: wykrywanie języka, podział tekstu na jednostki, stemming lub lematyzacja, korekta pisowni i identyfikowanie synonimów. Następnie budowana jest struktura danych (np. indeks odwrócony) umożliwiające wydajne wyszukiwanie.
- W systemach uczenia maszynowego model można uznać za pochodny od danych treningowych i uzyskany skutek zastosowania różnych funkcji wyodrębniania cech i analiz statystycznych. Gdy ten model jest stosowany do nowych danych wejściowych, dane wyjściowe są pochodne od wejściowych i modelu (a tym samym, pośrednio, od danych treningowych).
- Pamięć podręczna często zawiera agregację danych w formie, w jakiej mają być one wyświetlane w interfejsie użytkownika. Zapełnianie pamięci podręcznej wymaga więc wiedzy o tym, które pola są używane w interfejsie użytkownika. Zmiany w tym interfejsie mogą wymagać zaktualizowania definicji procesu zapełniania pamięci podręcznej i jej ponownego budowania.

Funkcje generujące indeksy pomocnicze są potrzebne tak często, że w wielu bazach stanowią jeden z podstawowych wbudowanych mechanizmów. Możesz je wywołać za pomocą samego wywołania `CREATE INDEX`. Jeśli chodzi o indeksy pełnotekstowe, podstawowe funkcje lingwistyczne dla popularnych języków mogą być wbudowane w bazę, jednak bardziej zaawansowane mechanizmy często wymagają specyficznych dla dziedziny modyfikacji. W uczeniu maszynowym inżynieria funkcji jest specyficzna dla aplikacji i często wymaga uwzględnienia szczegółowej wiedzy na temat interakcji z użytkownikami oraz środowiska pracy aplikacji [35].

Jeśli funkcja generująca pochodny zbiór danych nie jest standardową, szablonową funkcją (np. tworzącą indeks pomocniczy), potrzebny jest niestandardowy kod obsługi aspektów specyficznych dla aplikacji. To w obszarze tego niestandardowego kodu w wielu bazach pojawiają się trudności. Choć bazy relacyjne zwykle udostępniają wyzwalacze, procedury składowane i funkcje zdefiniowane przez użytkownika, które można wykorzystać do wykonywania kodu aplikacji w bazie, są one pewnego rodzaju dodatkiem do projektu baz (zob. punkt „Przesyłanie strumieni zdarzeń”).

## Oddzielenie kodu aplikacji od stanu

Bazy danych teoretycznie mogłyby być środowiskami wdrażania dowolnego kodu aplikacji (podobnie jak system operacyjny). Jednak w praktyce okazały się słabo dostosowane do tego celu. Nie są dobrze dopasowane do wymagań rozwoju nowoczesnych aplikacji — nie zapewniają zarządzania zależnościami i pakietami, kontroli wersji, stopniowych aktualizacji, możliwości modyfikowania, monitorowania, pomiarów, wywołań usług sieciowych i integracji z zewnętrznymi systemami.

Z kolei narzędzia do zarządzania wdrażaniem i klastrami, np. Mesos, YARN, Docker, Kubernetes i in., są zaprojektowane specjalnie na potrzeby uruchamiania kodu aplikacji. Dzięki temu, że mają dobrze robić jedną rzecz, potrafią wykonywać ją znacznie lepiej niż baza danych umożliwiającą wykonywanie zdefiniowanych przez użytkownika funkcji jako jeden z wielu mechanizmów.

Uważam, że sensowne jest, by niektóre części systemu specjalizowały się w trwałym składowaniu danych, a inne w uruchamianiu kodu aplikacji. Te dwa obszary mogą wchodzić ze sobą w interakcje, a przy tym pozostawać niezależne.

Obecnie większość aplikacji sieciowych jest instalowana jako usługi bezstanowe, w których dowolne żądania od użytkowników można kierować na dowolny serwer aplikacji, a serwer po odeśłaniu odpowiedzi zapomina wszystko na temat danego żądania. Ten styl wdrażania oprogramowania jest wygodny, ponieważ pozwala swobodnie dodawać i usuwać serwery. Jednak stan musi zostać gdzieś zapisany — zwykle w bazie. Obowiązuje trend nieumieszczania logiki aplikacji w bazie i nieumieszczania trwałego stanu w aplikacji [36]. Ludzie ze społeczności programowania funkcyjnego lubią żartować, że: „wierzą w rozdział Kościoła od państwa” [37]<sup>2</sup>.

W typowym modelu aplikacji sieciowych baza działa jak swoista modyfikowalna zmienna współużytkowana, dostępna synchronicznie w sieci. Aplikacja może wczytywać i aktualizować zmienną, a baza odpowiada za jej trwałość, zapewniając przy tym kontrolę współbieżności i odporność na błędy.

Jednak w większości języków programowania nie da się subskrybować zmian modyfikowalnej zmiennej. Można tylko okresowo ją wczytywać. Inaczej niż w arkuszu kalkulacyjnym jednostki wczytujące zmienną nie otrzymują powiadomień o zmianie jej wartości. Możesz zaimplementować takie powiadomienia we własnym kodzie (jest to tzw. *wzorzec obserwatora*), jednak większość języków nie ma wbudowanej obsługi tego wzorca.

Bazy „odziedziczyły” to pasywne traktowanie modyfikowalnych danych. Jeśli chcesz się dowiedzieć, czy zawartość bazy się zmieniła, często jedyną możliwością jest odpytywanie (czyli okresowe powtarzanie zapytania). Mechanizm subskrybowania powiadomień o zmianach dopiero zaczyna być wprowadzany (zob. punkt „Obsługa strumieni zmian za pomocą interfejsu API”).

## Przepływ danych — interakcje między zmianami stanu a kodem aplikacji

Myślenie o aplikacjach w kategoriach przepływu danych prowadzi do zmiany zależności między kodem aplikacji a zarządzaniem stanem. Zamiast traktować bazę jak pasywną zmienną, na której aplikacja operuje, należy myśleć w kategoriach interakcji i współdziałania stanu, zmian stanu oraz przetwarzającego je kodu. Kod aplikacji reaguje na zmianę stanu w jednym miejscu, wprowadzając zmiany stanu w innej lokalizacji.

Z tym sposobem myślenia zetknąłeś się w punkcie „Strumienie a bazy danych”, gdzie opisano traktowanie dziennika zmian w bazie jak strumienia zdarzeń, które można subskrybować. Systemy przekazywania komunikatów, np. oparte na aktorach (zob. punkt „Przepływ danych za pomocą

---

<sup>2</sup> W języku angielskim brzmi to *We believe in the separation of Church and state*, gdzie *Church* może oznaczać zarówno Kościół, jak i matematyka Alonzo Churcha, a *state* to albo państwo, albo stan. Alonzo Church opracował rachunek lambda, wczesny system przetwarzania będący podstawą większości funkcyjnych języków programowania. W rachunku lambda nie ma modyfikowalnego stanu (nie występują w nim zmienne, których wartość można nadpisywać), dlatego można stwierdzić, że modyfikowalny stan jest sprzeczny z pracami Churcha.

przekazywania komunikatów”), także reagują na zdarzenia. Już w latach 80. w modelu *przestrzeni krotek* badano przedstawianie obliczeń rozproszonych za pomocą procesów obserwujących zmiany stanu i reagujących na nie [38, 39].

Wspomniano już, że podobny proces zachodzi w bazie, gdy wskutek zmiany danych uruchamiany jest wyzwalacz lub gdy indeks pomocniczy jest aktualizowany, aby odzwierciedlić zmianę w indeksowanej tabeli. Podział bazy danych na komponenty polega na zastosowaniu tego podejścia do tworzenia pochodnych zbiorów danych poza główną bazą: w ramach pamięci podręcznej, indeksów wyszukiwania pełnotekstowego, uczenia maszynowego lub systemów analitycznych. W tym celu można się posłużyć przetwarzaniem strumieniowym i systemami obsługi komunikatów.

Należy pamiętać, że aktualizowanie danych pochodnych nie jest tym samym co asynchroniczne wykonywanie prac, do czego tradycyjnie projektowane są systemy obsługi komunikatów (zob. punkt „Porównanie dzienników z tradycyjną obsługą komunikatów”):

- W trakcie aktualizowania danych pochodnych kolejność zmian stanu jest często ważna (jeśli na podstawie dziennika zdarzeń wygenerowano kilka widoków, zdarzenia muszą być przetwarzane w tej samej kolejności, aby widoki pozostały spójne ze sobą). W punkcie „Potwierdzenia i ponowne dostarczanie” opisano, że wiele brokerów komunikatów nie zachowuje kolejności, gdy ponownie dostarcza komunikaty, dla których nie otrzymano potwierdzenia. Wykluczony jest też podwójny zapis (zob. punkt „Utrzymywanie synchronizacji systemów”).
- Odporność na błędy ma duży wpływ na dane pochodne. Już utrata jednego komunikatu powoduje, że pochodny zbiór danych trwale traci synchronizację ze źródłem danych. Zarówno dostarczanie komunikatów, jak i aktualizacje stanu pochodnego muszą przebiegać niezawodnie. Na przykład w wielu systemach opartych na aktorach stan aktorów i komunikatów jest domyślnie przechowywany w pamięci. Oznacza to utratę stanu po awarii maszyny, w której aktor działa.

Stabilna kolejność komunikatów i przetwarzanie komunikatów w sposób odporny na błędy to wysokie wymagania. Jednak ich spełnienie jest znacznie tańsze i zapewnia większą niezawodność operacyjną niż korzystanie z transakcji rozproszonych. Nowe systemy przetwarzania strumieniowego potrafią zapewniać gwarancje uporządkowania i niezawodności w dużej skali. Umożliwiają też uruchamianie kodu aplikacji działającego jako operator strumienia.

Kod aplikacji może wykonywać dowolne operacje, jakich wbudowane funkcje generujące z baz danych zwykle nie udostępniają. Operatory strumienia (podobnie jak narzędzia uniksowe łączone w potok) można złączać w duże systemy bazujące na przepływie danych. Każdy operator przyjmuje strumień zmian stanu jako dane wejściowe i generuje inne strumienie zmian stanu jako dane wyjściowe.

## Systemy przetwarzania strumieniowego i usługi

Modny obecnie styl rozwoju aplikacji polega na podziale mechanizmów na zbiór *usług*, które komunikują się między sobą za pomocą synchronicznych żądań sieciowych, np. przy użyciu interfejsu API REST (zob. punkt „Przepływ danych z użyciem usług: REST i RPC”). Zaletą architektury opartej na usługach w porównaniu z jedną monolityczną aplikacją jest przede wszystkim skalowalność w ramach organizacji dzięki luźnemu powiązaniu. Nad poszczególnymi usługami mogą pracować różne zespoły, co zmniejsza konieczność koordynowania działań (o ile usługi można wdrażać i aktualizować niezależnie od pozostałych).

Łączenie operatorów strumieni w systemy przepływu danych przypomina pod wieloma względami podejście oparte na mikrouslugach [40]. Jednak używany jest tu zupełnie inny mechanizm komunikacji — jednokierunkowe asynchroniczne strumienie komunikatów zamiast synchronicznych interakcji żądanie-odpowiedź.

Obok zalet wymienionych w punkcie „Przepływ danych za pomocą przekazywania komunikatów” (takich jak wyższa odporność na błędy) systemy przepływu danych mogą też zapewniać wyższą wydajność. Załóżmy, że klient kupuje produkt o cenie podanej w jednej walucie, ale płaci inną walutą. Aby przeprowadzić obliczenia, trzeba znać aktualny kurs wymiany tych walut. Taką operację można zaimplementować na dwa sposoby:

1. W podejściu z mikrouslugami kod przetwarzający zakup zapewne zgłosi zapytanie do usługi zwracającej kurs wymiany lub do bazy danych. W ten sposób uzyska aktualny kurs danych walut.
2. W podejściu opartym na przepływie danych kod przetwarzający zakup z góry zasubskrybuje strumień aktualizacji kursów wymiany i zapisze w lokalnej bazie aktualny kurs po każdej jego modyfikacji. W momencie przetwarzania transakcji kod będzie musiał tylko zgłosić zapytanie do lokalnej bazy.

W drugim podejściu synchroniczne żądanie sieciowe do innej usługi zostało zastąpione zapytaniem do lokalnej bazy (która może działać w tej samej maszynie, a nawet w tym samym procesie)<sup>3</sup>. Podejście oparte na przepływie danych jest nie tylko szybsze, ale też bardziej odporne na awarię innej usługi. Najszybsze i najbardziej niezawodne żądanie sieciowe to brak takiego żądania! Zamiast wywołań RPC stosowane jest złączanie strumieni dla zdarzeń zakupu i zdarzeń aktualizacji kursu wymiany (zob. punkt „Złączenia strumień-tabela (wzbogacanie strumieni)”).

Złączenia są zależne od czasu. Jeśli zdarzenie zakupu zostanie później ponownie przetworzone, kurs wymiany będzie inny. Jeżeli chcesz odtworzyć oryginalne dane wyjściowe, musisz pobrać historyczny kurs wymiany z czasu zakupu. Niezależnie od tego, czy kierujesz zapytania do usługi, czy subskrybujesz strumień aktualizacji kursów wymiany, musisz uwzględnić zależność od czasu (zob. punkt „Zależność od czasu w złączeniach”).

Subskrybowanie strumienia zmian (zamiast kierowania zapytań o bieżący stan, gdy jest to potrzebne) zbliża nas do modelu obliczeń przypominającego arkusz kalkulacyjny. Gdy jakiś fragment danych się zmienia, zależne od niego dane pochodne można szybko zaktualizować. Nadal trzeba rozwiązać wiele kwestii, np. związanych ze złączeniami zależnymi od czasu, jednak wierzę, że budowanie aplikacji z wykorzystaniem idei z obszaru przepływu czasu jest bardzo obiecującym kierunkiem prac.

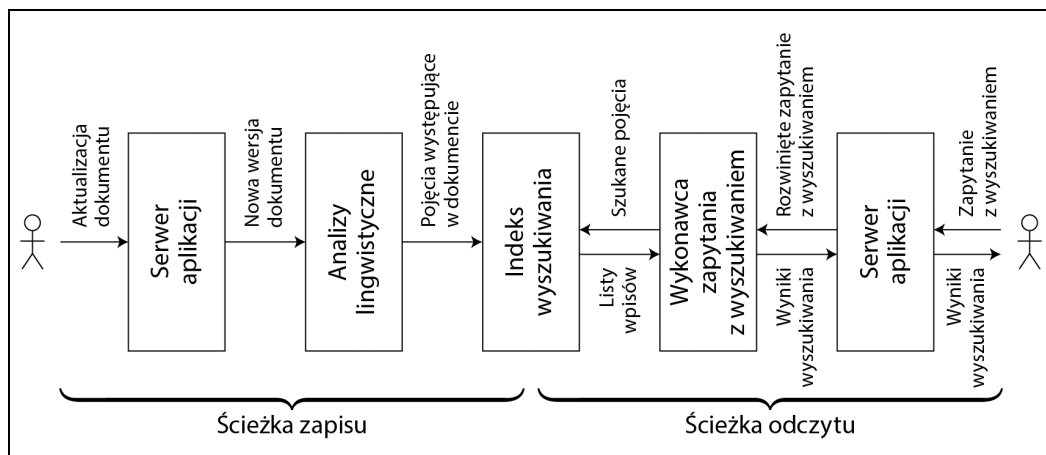
## Obserwowanie stanu pochodnego

Na abstrakcyjnym poziomie opisane we wcześniejszym punkcie systemy przepływu danych zapewniają proces tworzenia pochodnych zbiorów danych (np. indeksów wyszukiwania, widoków zmaterializowanych i modeli prognostycznych) i aktualizowania ich. Nazwijmy ten proces *ścieżką*

---

<sup>3</sup> W podejściu z mikrouslugami można uniknąć synchronicznych żądań sieciowych, zapisując kurs wymiany lokalnie w pamięci podręcznej w usłudze przetwarzającej zakup. Jednak aby pamięć podręczna zawierała aktualne dane, należy okresowo stosować odpytywanie i pobierać aktualne kursy wymiany lub, tak jak w podejściu opartym na przepływie danych, zasubskrybować strumień zmian.

zapisu. Za każdym razem, gdy jakaś porcja informacji jest zapisywana w systemie, może ona przejść przez wiele etapów przetwarzania wsadowego i strumieniowego. Ponadto ostatecznie wszystkie pochodne zbiory danych są aktualizowane na podstawie zapisanych danych. Na rysunku 12.1 pokazano przykład aktualizacji indeksu wyszukiwania.



Rysunek 12.1. W indeksie wyszukiwania zapisy (aktualizacje dokumentu) spotykają się z odczytami (zapytaniami)

Ale po co w ogóle tworzysz pochodny zbiór danych? Zapewne dlatego, że chcesz później ponownie kierować do niego zapytania. Jest to *ścieżka odczytu*. Gdy obsługujesz żądania użytkownika, wczytujesz informacje z pochodnego zbioru danych, zapewne przetwarzasz wyniki i generujesz odpowiedź.

Ścieżki zapisu i odczytu razem obejmują całą podróż danych od momentu ich zarejestrowania do chwili ich wykorzystania (prawdopodobnie przez innego człowieka). Ścieżka zapisu to część podróży, która jest wstępnie przetwarzana. Wykonuje się ją zaraz po nadejściu danych niezależnie od tego, czy ktoś zażądał ich wyświetlenia. Jeśli znasz funkcyjne języki programowania, może zauważyłeś, że ścieżka zapisu jest podobna do wartościowania zachłannego, a ścieżka odczytu — do wartościowania leniwego.

Pochodny zbiór danych to miejsce, w którym ścieżki zapisu i odczytu się spotykają. Widać to na rysunku 12.1. Ten zbiór danych to wynik kompromisu między ilością pracy, jaką trzeba wykonać w czasie zapisu, a obciążeniem zadaniami w czasie odczytu.

## Widoki zmaterializowane i pamięć podręczna

Dobrym przykładem są tu indeksy wyszukiwania pełnotekstowego. Ścieżka zapisu aktualizuje indeks, a ścieżka odczytu przeszukuje go pod kątem słów kluczowych. Zarówno odczyty, jak i zapisy wymagają wykonania pracy. Zapisy muszą zaktualizować w indeksie wpisy dotyczące wszystkich pojęć występujących w danym dokumencie. W ramach odczytu trzeba wyszukać wszystkie słowa z zapytania i na podstawie operacji logicznych znaleźć dokumenty obejmujące *wszystkie* słowa z zapytania (operator AND) lub *dowolny* synonim każdego ze słów (operator OR).

Jeśli nie masz indeksu, zapytanie z wyszukiwaniem musi przeskanować wszystkie dokumenty (tak jak w poleceniu `grep`). Jest to bardzo kosztowne, gdy dokumentów jest dużo. Brak indeksu oznacza mniej pracy w ścieżce zapisu (nie trzeba aktualizować indeksu), ale znacznie więcej w ścieżce odczytu.



Z drugiej strony możesz wyobrazić sobie wstępne obliczanie wyników wyszukiwania z wszystkich możliwych zapytań. Wtedy ilość pracy w ścieżce odczytu jest mniejsza. Nie trzeba stosować operacji logicznych, a jedynie znaleźć wyniki zapytania i zwrócić je. Jednak ścieżka zapisu byłaby wtedy dużo bardziej kosztowna. Zbiór możliwych zapytań z wyszukiwaniem jest nieskończony. Dlatego wstępne obliczenie wszystkich możliwych wyników wyszukiwania wymagałoby nieskończonej ilości czasu i miejsca. Takie rozwiązanie raczej się nie sprawdzi<sup>4</sup>.

Inna możliwość to wstępne obliczanie wyników wyszukiwania tylko dla stałego zbioru najczęściej zgłaszanych zapytań, aby można je było szybko obsługiwać bez konieczności używania indeksu. Nietypowe zapytania nadal są wtedy obsługiwane z użyciem indeksu. Takie rozwiązanie zwykle jest nazywane *pamięcią podręczną* dla popularnych zapytań, choć można je nazwać także widokiem zmaterializowanym, ponieważ konieczna jest aktualizacja po pojawieniu się nowych dokumentów, które należy uwzględnić w wynikach jednego z popularnych zapytań.

Z tego przykładu wynika, że indeks nie jest jedyną możliwą granicą między ścieżkami zapisu i odczytu. Możliwe jest też zapisywanie wyników popularnych wyszukiwań w pamięci podręcznej. Ponadto gdy dokumentów jest niewiele, można stosować skanowanie (bez indeksu) podobne jak w poleceniu `grep`. W tym kontekście zadanie pamięci podręcznej, indeksów i widoków zmaterializowanych jest proste — przesuwają one granicę między ścieżkami zapisu i odczytu. Pozwalają wykonać więcej pracy w ścieżce zapisu (za pomocą wstępnego obliczania wyników) w celu uniknięcia wysiłku w ścieżce odczytu.

Przesuwanie granicy między pracą wykonywaną w ścieżkach zapisu i odczytu było tematem dotyczącego Twittera przykładu z początku tej książki (punkt „Opis obciążenia”). W tamtym przykładzie zobaczyłeś też, że granice między ścieżkami zapisu i odczytu można wyznaczać inaczej dla gwiazd, a inaczej dla zwykłych użytkowników. Po 500 stronach wróciliśmy do punktu wyjścia!

## Klienty stanowe z możliwością działania w trybie offline

Kwestia granicy między ścieżkami zapisu i odczytu jest ciekawa, ponieważ można dyskutować o przesuwaniu tej granicy i analizować, jakie są tego praktyczne skutki. Przyjrzyjmy się teraz temu zagadnieniu w innym kontekście.

Olbrzymia popularność aplikacji sieciowych w dwóch ostatnich dekadach doprowadziła do przyjęcia pewnych założeń na temat rozwoju oprogramowania, które to założenia łatwo jest brać za pewnik. Przede wszystkim model klient-serwer (w którym klienci są zwykle bezstanowe, a za dane odpowiadają serwery) jest tak popularny, że prawie zapomniano o istnieniu innych rozwiązań. Jednak technologia się rozwija, dlatego uważam, że ważne jest, by od czasu do czasu zakwestionować status quo.

W przeszłości przeglądarki internetowe były bezstanowymi klientami, potrafiącymi wykonywać przydatne operacje tylko wtedy, gdy dostępne było połączenie internetowe (w trybie offline niemal jedyną możliwą operacją było przewijanie stron wczytanych wcześniej w trybie online). Jednak wprowadzone niedawno jednostronicowe aplikacje sieciowe w JavaScriptcie mają wiele możliwości związanych ze stanem. Obsługują np. interakcję z interfejsem użytkownika po stronie klienta i trwają

---

<sup>4</sup> Bardziej na poważnie warto zauważyć, że zbiór różnych zapytań z wyszukiwaniem o niepustych wynikach jest skończony (przy założeniu, że korpus też jest skończony). Jednak ten zbiór rośnie wykładniczo wraz z rozrastaniem się korpusu, co też jest złą informacją.

pamięć lokalną w przeglądarce. Podobnie aplikacje mobilne mogą przechowywać dużą część stanu w urządzeniu i nie wymagają wymiany komunikatów z serwerem dla prawie każdej interakcji z użytkownikiem.

Zmiany możliwości doprowadziły do ponownego ożywienia zainteresowania aplikacjami działającymi *głównie w trybie offline*. Wykonują one jak najwięcej zadań za pomocą lokalnej bazy z danego urządzenia (bez konieczności łączenia się z internetem) i synchronizują stan ze zdalnymi serwerami w tle, gdy połączenie z internetem jest dostępne [42]. Ponieważ urządzenia mobilne często korzystają z powolnego i zawodnego połączenia internetowego przez sieć komórkową, dużą zaletą dla użytkowników jest to, że ich interfejs użytkownika nie musi czekać na synchroniczne żądania sieciowe, a aplikacje działają głównie w trybie offline (zob. punkt „Klienty działające w trybie offline”).

Po rezygnacji z założenia, że klienty bezstanowe komunikują się z centralną bazą, i zaakceptowaniu przechowywania stanu w urządzeniach użytkowników otwiera się całe mnóstwo nowych możliwości. Przede wszystkim można traktować stan w urządzeniu jak *pamięć podręczną stanu z serwera*. Piksele na ekranie są zmaterializowanym widokiem obiektów modelu z aplikacji klienckiej. Obiekty modelu są z kolei lokalną repliką stanu ze zdalnego centrum danych [27].

### Przekazywanie zmian stanu do klientów

W typowej witrynie internetowej po wczytaniu strony w przeglądarce i przesłaniu danych na serwer przeglądarka wykrywa modyfikacje dopiero po odświeżeniu strony. Przeglądarka wczytuje dane tylko w jednym momencie, zakładając, że są one statyczne; nie subskrybuje natomiast aktualizacji z serwera. Dlatego stan w urządzeniu to niezmienna pamięć podręczna, która nie jest aktualizowana, chyba że bezpośrednio zażądaś zmian za pomocą odpytywania. Oparte na HTTP protokoły subskrypcji kanałów informacyjnych, np. RSS, korzystają z prostego odpytywania.

W nowszych protokołach nastąpiło wyjście poza podstawowy wzorzec żądanie-odpowiedź używany w HTTP. Zdarzenia przesyłane przez serwer (za pomocą interfejsu API EventSource) i technologia WebSocket zapewniają kanały komunikacyjne, dzięki którym przeglądarka może utrzymywać otwarte połączenie TCP z serwerem. Serwer może wtedy aktywnie przysyłać komunikaty do przeglądarki, o ile połączenie jest utrzymywane. Serwer może więc aktywnie informować klienta użytkownika końcowego o zmianach w lokalnie przechowywanym stanie. Dzięki temu stan po stronie klienta jest w mniejszym stopniu nieaktualny.

W modelu ścieżek zapisu i odczytu aktywne przysyłanie zmian w stanie do urządzeń klienta oznacza rozszerzenie ścieżki zapisu aż do użytkownika końcowego. Gdy klient jest inicjowany, musi za pomocą ścieżki dostępu pobrać początkowy stan. Jednak później może polegać na przesyłanym przez serwer strumieniu zmian stanu. Pomysły opisane w związku z przetwarzaniem strumieniowym i komunikatami dotyczą nie tylko centrum danych. Można je rozciągnąć na urządzenia użytkownika końcowego [43].

Urządzenia przez część czasu są w trybie offline i nie mogą wtedy otrzymywać powiadomień o zmianach stanu od serwera. Jednak ten problem został już rozwiązany. W punkcie „Pozycje odnotowane przez konsumentów” opisano, że konsument brokera komunikatów opartego na dziennikach może ponownie nawiązać połączenie po awarii lub rozłączeniu się i upewnić się, iż nie pominął

żadnych komunikatów przesłanych w czasie, gdy połączenie było nieaktywne. Ta sama technika sprawdza się w przypadku pojedynczych użytkowników, gdzie każde urządzenie to niewielki subskrybent małego strumienia zdarzeń.

### **Strumienie zdarzeń łączące punkty końcowe**

Nowe narzędzia do budowania stanowych klientów i interfejsów użytkownika, np. język Elm [30] i łańcuch programów React, Flux, Redux z Facebooka [44], już zarządzają wewnętrznym stanem po stronie klienta, subskrybując strumień zdarzeń reprezentujących dane wejściowe od użytkownika lub odpowiedzi z serwera. Te rozwiązania działają podobnie jak event sourcing (zob. punkt „Event sourcing”).

Bardzo naturalne jest rozwinięcie tego modelu programowania w taki sposób, by umożliwić serwerowi przesyłanie zdarzeń zmiany stanu do potoku zdarzeń po stronie klienta. Zmiany stanu mogą więc przepływać ścieżką zapisu między punktami końcowymi: od interakcji w jednym urządzeniu, która wywołała zmianę stanu, przez dzienniki zdarzeń, kilka systemów z danymi pochodnymi i systemów przetwarzania strumieniowego, aż do interfejsu użytkownika w innym urządzeniu, gdzie stan jest obserwowany. Te zmiany stanu można przekazywać ze stosunkowo niskim opóźnieniem (np. poniżej sekundy od jednego punktu końcowego do drugiego).

W niektórych aplikacjach, np. w komunikatorach i grach internetowych, działa już tego typu architektura „czasu rzeczywistego” (w sensie interakcji o niskim opóźnieniu, a nie rozwiązań z punktu „Gwarantuje czas zwracania odpowiedzi”). Dlaczego jednak nie budować wszystkich aplikacji w ten sposób?

Trudność polega na tym, że założenie o bezstanowych klientach i interakcjach żądanie-odpowiedź jest bardzo mocno wbudowane w bazy danych, biblioteki, platformy i protokoły. Wiele magazynów danych obsługuje operacje odczytu i zapisu, gdzie żądanie powoduje zwrócenie jednej odpowiedzi. Znacznie mniej rozwiązań umożliwia subskrybowanie zmian (czyli żądań prowadzących do zwracania później strumienia odpowiedzi; zob. punkt „Obsługa strumieni zmian za pomocą interfejsu API”).

Aby wydłużyć ścieżkę zapisu aż do użytkownika końcowego, trzeba całkowicie zmienić sposób myślenia o budowaniu wielu systemów i przejść od interakcji żądanie-odpowiedź w kierunku przepływu danych w modelu publikuj-subskrybuj [27]. Uważam, że korzyści płynące z szybciej reagujących interfejsów użytkownika i lepszej obsługi trybu offline są warte tego wysiłku. Jeśli projektujesz systemy danych, mam nadzieję, że będziesz pamiętać o możliwości subskrybowania zmian (obok samych zapytań o aktualny stan).

### **Odczyty też są zdarzeniami**

Opisano, że gdy system przetwarzania strumieniowego zapisuje dane pochodne w ich magazynie (bazie, pamięci podręcznej lub indeksie) i gdy żądania użytkownika kierują zapytania do tego magazynu, ten ostatni działa jak granica między ścieżkami zapisu i odczytu. Taki magazyn umożliwia zgłaszanie zapytań wymagających odczytu z dostępem swobodnym, które normalnie wymagałyby skanowania całego dziennika zdarzeń.

W wielu sytuacjach magazyn danych jest odrębny od systemu przetwarzania strumieniowego. Pamiętaj jednak, że taki system musi utrzymywać stan na potrzeby agregacji i złączeń (zob. punkt „Złączanie strumieni”). Ten stan jest zwykle ukryty w systemie przetwarzania strumieniowego,

jednak niektóre platformy umożliwiają zewnętrznym klientom kierowanie zapytań o stan [45]. Taki system działa wtedy jak prosta baza.

Chciałbym rozwinąć ten pomysł. Zgodnie z wcześniejszymi opisami zapisy wprowadzane w magazynie przechodzą przez dziennik zdarzeń. Z kolei odczyty to tymczasowe żądania sieciowe kierowane bezpośrednio do węzłów przechowujących potrzebne dane. Jest to sensowny projekt, ale możliwe są też inne rozwiązania. Żądania odczytu można także reprezentować jako strumienie zdarzeń i przysyłać zarówno zdarzenia odczytu, jak i zdarzenia zapisu do systemu przetwarzania strumieniowego. Ten system powinien reagować na zdarzenia odczytu, przekazując wynik odczytu do strumienia wyjściowego [46].

Gdy i zapisy, i odczyty są reprezentowane jako zdarzenia oraz kierowane w celu obsługi do tego samego operatora strumienia, wykonywane jest złączenie strumień-tabela między strumieniem zapytań wymagających odczytu a bazą. Zdarzenie odczytu należy przesłać do partycji bazy przechowującej potrzebne dane (zob. punkt „Trasowanie żądań”). Podobnie systemy przetwarzania wsadowego i strumieniowego w trakcie łączenia muszą łączyć dane wejściowe na podstawie tego samego klucza (zob. punkt „Złączanie i grupowanie na etapie redukcji”).

Te analogie między obsługą żądań a przeprowadzaniem złączeń są czymś fundamentalnym [47]. Jednorazowe żądanie odczytu jest przekazywane do operatora złączenia i natychmiast zapominać. Żądanie subskrypcji to trwałe złączenie obejmujące przeszłe i przyszłe zdarzenia z obu stron złączenia.

Rejestrowanie dziennika zdarzeń odczytu może też przynieść korzyści łączące się ze śledzeniem związków przyczynowych i pochodzeniem danych w systemie. Można wtedy odtworzyć, co użytkownik widział przed podjęciem określonej decyzji. Na przykład w sklepie internetowym wyświetlane użytkownikowi prognozowana data dostawy i dostępność produktu w magazynie zapewne wpłyną na decyzję o zakupie [4]. Aby przeanalizować takie zależności, trzeba zarejestrować wynik zapytania użytkownika o dostawę i dostępność produktu.

Zapis zdarzeń odczytu w trwałej pamięci usprawnia więc śledzenie związków przyczynowych (zob. punkt „Porządkowanie zdarzeń w celu uchwycenia przyczynowości”). Powoduje jednak dodatkowe koszty związane z pamięcią i operacjami wejścia-wyjścia. Optymalizowanie takich systemów w celu ograniczenia kosztów jest wciąż nierozwiązanym problemem badawczym [2]. Jeśli jednak i tak rejestrujesz zdarzenia odczytu w celach operacyjnych (jako efekt uboczny przetwarzania żądania), użycie dziennika jako źródła żądań nie wymaga wielkich zmian.

## Przetwarzanie danych z wielu partycji

W przypadku zapytań dotyczących tylko jednej partycji przysyłanie zapytań do strumienia i pobieranie strumienia odpowiedzi jest zapewne przesadą. Jednak opisany pomysł otwiera możliwość rozproszonego wykonywania złożonych zapytań, które wymagają łączenia danych z kilku partycji. Pozwala to wykorzystać infrastrukturę trasowania komunikatów, podziału na partycję i złączania, która jest już udostępniana przez system przetwarzania strumieniowego.

Mechanizm rozproszonych wywołań RPC z systemu Storm obsługuje ten wzorzec użytkownika (zob. punkt „Przekazywanie komunikatów i wywołania RPC”). W ten sposób określano np. liczbę osób, które zobaczyły dany adres URL na Twitterze (odpowiada ona sumie zbiorów obserwatorów ludzi, którzy zamieścili tweet z tym adresem) [48]. Ponieważ zbiór użytkowników Twittera jest podzielony na partycje, te obliczenia wymagają łączenia wyników z wielu partycji.

Inny przykład wykorzystania tego wzorca dotyczy zapobiegania oszustwom. Aby ocenić ryzyko, że dane zdarzenie zakupu jest próbą oszustwa, można sprawdzić reputację adresu IP użytkownika, adresu e-mail, adresu do faktury, adresu dostawy itd. Każda baza z reputacją jest podzielona na partycje, dlatego zebranie wartości dla konkretnego zdarzenia zakupu wymaga sekwencji złączeń między podzielonymi w odmienny sposób zbiorami danych [49].

Wewnętrzne grafy wykonywania zapytań w bazach MP mają podobne cechy (zob. punkt „Porównanie Hadoopa z rozproszonymi bazami danych”). Jeśli chcesz wykonywać złączenia z udziałem wielu partycji, zapewne łatwiej będzie zastosować bazę udostępniającą taką funkcję, niż implementować ten mechanizm za pomocą systemu przetwarzania strumieniowego. Jednak traktowanie zapytań jak strumieni umożliwia implementowanie działających na dużą skalę aplikacji, które przekraczają ograniczenia typowe dla standardowych gotowych rozwiązań.

## Dążenie do poprawności

Gdy stosujesz bezstanowe usługi, które tylko wczytują dane, błędy nie stanowią dużego problemu. Możesz naprawić błąd i ponownie uruchomić usługę, a wszystko wróci do normalności. Systemy stanowe (np. bazy) są bardziej skomplikowane. Są projektowane tak, by zapamiętywały dane na zawsze (mniej więcej). Dlatego także skutki błędów mogą być wieczne. Oznacza to, że bazy wymagają bardziej starannego przemyślenia [50].

Celem jest budowanie aplikacji, które są niezawodne i *poprawne* (czyli programów o dobrze zdefiniowanej i zrozumiałej semantyce — nawet w obliczu różnych błędów). Przez ok. 40 lat do tworzenia prawidłowych aplikacji wykorzystywano z wyboru cechy transakcji — atomowość, izolację i trwałość (zob. rozdział 7.). Jednak te podstawy są słabsze, niż może się wydawać. Pomyśl np. o wątpliwościach występujących na niskich poziomach izolacji (zob. punkt „Niskie poziomy izolacji”).

W niektórych obszarach całkowicie zrezygnowano z transakcji na rzecz modeli, które oferują lepszą wydajność i skalowalność, ale kosztem bardziej skomplikowanej semantyki (zob. np. punkt „Replikacja bez lidera”). Często mówi się o *spójności*, jest ona jednak słabo zdefiniowana (zob. punkt „Spójność” i rozdział 9.). Część osób uważa, że powinniśmy „pogodzić się z niską spójnością” w zamian za wyższą dostępność. Jednak te osoby nie wiedzą, co to będzie oznaczać w praktyce.

Jak na tak ważne zagadnienie nasze zrozumienie go i metody inżynierskie są na zaskakująco niskim poziomie. Na przykład bardzo trudno jest ustalić, czy można bezpiecznie uruchomić konkretną aplikację z określonym poziomem izolacji transakcji lub daną konfiguracją replikacji [51, 52]. Proste rozwiązania przy niskim poziomie współbieżności i bez awarii często wydają się działać poprawnie, jednak w bardziej wymagających okolicznościach ujawniają wiele subtelnych usterek.

Na przykład eksperymenty Kyle’a Kingsbury’ego z biblioteką Jepsen [53] ujawniły wyraźne rozbieżności między deklarowanymi gwarancjami bezpieczeństwa produktu a jego rzeczywistym zachowaniem w obecności problemów z siecią i awarii. Nawet gdy produkty infrastrukturalne, np. bazy, są wolne od problemów, kod aplikacji i tak musi poprawnie korzystać z udostępnianych przez nie funkcji. Łatwo jednak popełnić błędy, gdy konfiguracja jest trudna do zrozumienia (jak to się dzieje na niskich poziomach izolacji, przy konfigurowaniu quorum itd.).

Jeśli aplikacja jest odporna na okazjonalne uszkodzenie lub zgubienie danych w nieprzewidywalny sposób, praca jest znacznie łatwiejsza. Może wtedy ująć Ci na sucho trzymanie kciuków i liczenie na cud. Jeżeli jednak potrzebujesz mocniejszych gwarancji poprawności, uznanymi rozwiązaniami są sekwencyjność i zatwierdzanie atomowe. Są one jednak związane z kosztami: zwykle działają tylko w ramach jednego centrum danych (co wyklucza stosowanie geograficznie rozproszonych architektur) oraz ograniczają skalowalność i odporność na błędy.

Choć tradycyjne podejście oparte na transakcjach nie odejdzie w przeszłość, uważam, że nie jest to ostatnie słowo w dziedzinie zapewniania poprawności aplikacji i ich odporności na błędy. W tym punkcie proponuję kilka sposobów myślenia o poprawności a kontekście architektur przepływu danych.

## Zasada punktów końcowych dla baz danych

Samo to, że aplikacja korzysta z systemu danych zapewniającego stosunkowo silne cechy z obszaru bezpieczeństwa (np. sekwencyjne transakcje), nie gwarantuje, iż program będzie wolny od utraty lub uszkodzenia danych. Na przykład jeśli w aplikacji występuje błąd powodujący zapis nieprawidłowych danych lub usuwanie danych z bazy, sekwencyjne transakcje Cię przed tym nie ochronią.

Ten przykład może się wydawać śmieszny, jednak warto potraktować go poważnie. Błędy w aplikacjach się zdarzają, a ludzie popełniają pomyłki. Posłużyłem się tym przykładem w punkcie „Stan, strumień i niemodyfikowalność”, argumentując na rzecz niemodyfikowalnych danych, które można wyłącznie dołączać. Łatwiej jest przywrócić stan po błędach, jeśli wyeliminuje się ryzyko zniszczenia poprawnych danych przez nieprawidłowy kod.

Choć niemodyfikowalność jest użyteczna, sama w sobie nie stanowi rozwiązania. Przyjrzyj się teraz bardziej subtelnemu scenariuszowi uszkodzenia danych.

### Wykonywanie operacji tylko raz

W punkcie „Odporność na błędy” pojawiła się semantyka *dokładnie jednokrotnego wykonania* (lub *praktycznie jednokrotnego*). Po wystąpieniu błędu w trakcie przetwarzania można albo zrezygnować (usunąć komunikat, czyli spowodować utratę danych), albo ponowić próbę. Następną próbą grozi tym, że jeśli za pierwszym razem operacja się powiodła, ale nadawca się o tym nie dowiedział, komunikat zostanie przetworzony dwukrotnie.

Ponowne przetwarzanie jest odmianą uszkodzenia danych. Niepożądane jest dwukrotne obciążanie klienta kosztami za tę samą usługę (naliczanie zbyt wysokiej opłaty) lub dwukrotne zwiększanie licznika (przeszacowanie jakiegoś pomiaru). W tym kontekście semantyka *dokładnie raz* oznacza, że obliczenia należy tak zaplanować, by ostateczny efekt był taki sam jak w sytuacji, gdy problemy nie wystąpiły — nawet jeśli z powodu błędu operację ponowiono. Wcześniej opisano kilka sposobów na osiągnięcie tego celu.

Jednym z najskuteczniejszych rozwiązań jest zapewnienie *idempotencji* operacji (zob. punkt „Idempotencja”). Gwarantuje to, że efekt jest taki sam niezależnie od liczby powtórzeń danego zadania. Jednak przekształcenie w taki sposób operacji, która z natury nie jest idempotentna, wymaga wysiłku i staranności. Możliwe, że będziesz musiał przechowywać dodatkowe metadane (np. zbiór identyfikatorów operacji, które aktualizowały daną wartość) i stosować odgradzanie przy przełączaniu awaryjnym jednego węzła na inny (zob. punkt „Lider i blokady”).

## Zapobieganie duplikatom

Ten sam wzorzec wymogu zapobiegania duplikatom powtarza się też w wielu innych miejscach oprócz przetwarzania strumieniowego. Na przykład w protokole TCP używane są numery porządkowe pakietów, aby odpowiednio uporządkować te ostatnie po stronie odbiorcy i móc stwierdzić, czy pakiety nie zostały utracone lub zduplikowane w sieci. Utracone pakiety są ponownie przesyłane, a duplikaty usuwane przez stos narzędzi z protokołu TPC przed przekazaniem danych aplikacji.

Jednak zapobieganie duplikatom sprawdza się tylko w kontekście jednego połączenia TCP. Wyobraź sobie, że łączy ono klienta z bazą i wykonywana jest transakcja z listingu 12.1. W wielu bazach transakcja jest powiązana z połączeniem klienta. Jeśli klient przesyła kilka zapytań, baza „wie”, że należą one do tej samej transakcji, ponieważ zostały przesłane tym samym połączeniem TCP. Jeżeli w sieci wystąpią zakłócenia i przekroczony zostanie limit czasu po przesłaniu przez klienta polecenia COMMIT, ale przed zwróceniem odpowiedzi przez serwer bazodanowy, klient nie będzie wiedział, czy transakcja została zatwierdzona, czy anulowana (rysunek 8.1).

*Listing 12.1. Nieidempotentny przelew pieniędzy z jednego konta na drugie*

```
BEGIN TRANSACTION;  
UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;  
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;  
COMMIT;
```

Klient może jeszcze raz połączyć się z bazą i ponowić transakcję, ale wtedy nie obejmuje go zapobieganie duplikatom zapewniane przez protokół TCP. Ponieważ transakcja z listingu 12.1 nie jest idempotentna, może się zdarzyć, że przelane zostanie 22 zł zamiast oczekiwanych 11 zł. Dlatego choć kod z listingu 12.1 to standardowy przykład zapewniania atomowości transakcji, nie jest to poprawne rozwiązanie i rzeczywiste banki go nie stosują [3].

Protokoły zatwierdzania dwuetapowego (zob. punkt „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”) naruszają odwzorowanie 1:1 między połączeniem TCP a transakcją, ponieważ muszą umożliwić koordynatorowi transakcji ponowne połączenie się z bazą po awarii sieci i nakazanie jej zatwierdzenia lub anulowania wątplivej transakcji. Czy to wystarczy, aby zagwarantować, że transakcja zostanie wykonana tylko raz? Niestety nie.

Nawet jeśli zdołasz zapobiec duplikowaniu transakcji na drodze między klientem a serwerem bazodanowym, nadal trzeba uwzględnić sieć między urządzeniem użytkownika końcowego a serwerem aplikacji. Jeśli klientem użytkownika jest przeglądarka internetowa, zapewne używa ona żądań HTTP POST do przesyłania instrukcji na serwer. Użytkownik prawdopodobnie korzysta z niestabilnego komórkowego połączenia internetowego, dlatego sygnał staje się za słaby, zanim urządzenie zdoła odebrać odpowiedź z serwera.

W takim scenariuszu użytkownik zwykle widzi komunikat o błędzie i może ręcznie ponowić próbę. Przeglądarka wyświetla ostrzeżenie: „Czy na pewno chcesz jeszcze raz przesłać formularz?”, a użytkownik potwierdza, ponieważ chce, aby operacja została wykonana. Wzorzec *Post/Redirect/Get* [54] pozwala uniknąć komunikatów ostrzegawczych w trakcie normalnego działania aplikacji, ale nie jest pomocny, gdy nastąpi przekroczenie limitu czasu dla żądania POST. Dla serwera WWW ponowienie próby to odrębne żądanie, a dla bazy jest to odrębna transakcja. Dlatego standardowe mechanizmy zapobiegania duplikatom nie zadziałają.

## Identyfikatory operacji

Aby operacja była idempotentna w ramach kilku etapów komunikacji sieciowej, nie wystarczy polegać na mechanizmie obsługi transakcji zapewnianym przez bazę. Trzeba uwzględnić przepływ żądań *między punktami końcowymi*.

Możesz np. wygenerować unikatowy identyfikator operacji (taki jak UUID) i umieścić go jako ukryte pole formularza w aplikacji klienckiej. Możesz też obliczyć skrót wszystkich istotnych pól formularza, aby uzyskać identyfikator operacji [3]. Jeśli przeglądarka dwukrotnie prześle żądanie POST, oba żądania będą miały ten sam identyfikator operacji. Ten identyfikator można przekazywać aż do bazy danych i upewniać się tam, że operacja o danym identyfikatorze jest wykonywana tylko raz. Ilustruje to listing 12.2.

*Listing 12.2. Zapobieganie zduplikowanym żądaniom za pomocą unikatowego identyfikatora*

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;

INSERT INTO requests
  (request_id, from_account, to_account, amount)
VALUES ('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);

UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;

COMMIT;
```

Na listingu 12.2 wykorzystano więzy unikatowości dotyczące kolumny `request_id`. Jeśli transakcja spróbuje wstawić identyfikator, który już istnieje, polecenie `INSERT` zakończy się niepowodzeniem i transakcja zostanie anulowana, co zapobiega dwukrotnemu jej wykonaniu. Bazy relacyjne zwykle potrafią poprawnie przestrzegać więzów unikatowości — nawet na niskich poziomach izolacji (natomiast model „sprawdź, a potem wstaw” używany w kodzie aplikacji może zawieść, gdy stosowany jest niesekwencyjny poziom izolacji; opisano to w punkcie „Zapis zniekształcający i fantomy”).

Obok zapobiegania zduplikowanym żądaniom tabela `requests` z listingu 12.2 działa jak dziennik zdarzeń i stanowi krok w kierunku techniki event sourcing (zob. punkt „Event sourcing”). Aktualizacje stanu konta nie muszą odbywać się w tej samej transakcji, która wstawiła zdarzenie, ponieważ są nadmiarowe i można je otrzymać na podstawie zdarzenia żądania w konsumencie z dalszego etapu przetwarzania. Warunkiem jest to, by zdarzenie było przetwarzane dokładnie raz, co także można wymusić za pomocą identyfikatora żądania.

## Zasada punktów końcowych

Scenariusz zapobiegania zduplikowanym transakcjom jest tylko jednym przykładem ogólniejszej *zasady punktów końcowych* (ang. *end-to-end argument*), przedstawionej przez Saltzera, Reeda i Clarka w 1984 r. [55]:

Dana funkcja może zostać w pełni i poprawnie zaimplementowana wyłącznie dzięki znajomości i pomocy aplikacji działającej w punktach końcowych systemu komunikacji. Dlatego udostępnienie omawianej funkcji jako mechanizmu samego systemu komunikacji jest niemożliwe. Czasem niekompletna wersja funkcji oferowana przez system komunikacji może być przydatna do poprawy wydajności.



W przykładzie *omawianą funkcją* jest zapobieganie duplikatom. Zobaczyłeś, że protokół TCP zapobiega zduplikowanym pakietom na poziomie połączenia TCP, a niektóre systemy przetwarzania strumieniowego zapewniają semantykę dokładnie jednokrotnego wykonania na poziomie przetwarzania komunikatów. To jednak nie wystarcza, aby uniemożliwić użytkownikowi przesłanie zduplikowanego żądania, jeśli pierwsze żądanie przekroczy limit czasu. Protokół TCP, transakcje w bazie i systemy przetwarzania strumieniowego same w sobie nie mogą całkowicie wyeliminować duplikatów. Rozwiązanie tego problemu wymaga rozwiązania na drodze od jednego punktu końcowego do drugiego. Tym rozwiązaniem jest identyfikator transakcji przekazywany od klienta użytkownika końcowego do bazy danych.

Zasada punktów końcowych dotyczy także sprawdzania integralności danych. Sumy kontrolne wbudowane w technologie Ethernet, TCP i TLS pozwalają wykryć uszkodzenie pakietów w sieci. Nie pozwalają jednak dostrzec problemów spowodowanych błędami w oprogramowaniu po obu stronach połączenia sieciowego lub usterki dysków, na których przechowywane są dane. Jeśli chcesz wykrywać wszystkie źródła uszkodzenia danych, musisz też przekazywać sumy kontrolne między punktami końcowymi.

Podobnie jest z szyfrowaniem [55]. Hasło w domowej sieci Wi-Fi chroni przed osobami podsłuchującymi komunikację w tej sieci, ale już nie przed napastnikami działającymi w internecie. Protokół TLS/SSL używany między klientem a serwerem zabezpiecza przed napastnikami z sieci, ale nie chroni przed włamaniem na serwer. Tylko szyfrowanie i uwierzytelnianie między punktami końcowymi może zabezpieczać przed wymienionymi atakami.

Choć niskopoziomowe mechanizmy (zapobieganie duplikatom w protokole TCP, sumy kontrolne w Ethernetie, szyfrowanie sieci Wi-Fi) nie mogą same w sobie zapewnić pożądanych mechanizmów między punktami końcowymi, są przydatne, ponieważ zmniejszają prawdopodobieństwo wystąpienia problemów na wyższych poziomach. Na przykład żądania HTTP często byłyby uszkodzane, gdyby protokół TCP nie zapewniał prawidłowej kolejności pakietów. Trzeba jedynie pamiętać, że niskopoziomowe funkcje zapewniania niezawodności nie wystarczają do uzyskania poprawności między punktami końcowymi.

## Stosowanie perspektywy punktów końcowych w systemach danych

To prowadzi do mojej pierwotnej tezy — samo to, że aplikacja używa systemu danych zapewniającego stosunkowo silne cechy z obszaru bezpieczeństwa (np. transakcje sekwencyjne), nie daje gwarancji ochrony przed utratą lub uszkodzeniem danych. Także w samej aplikacji trzeba stosować mechanizmy działające na poziomie punktów końcowych (np. zapobieganie duplikatom).

Jest to niekorzystne, ponieważ budowanie mechanizmów zapewniania odporności na błędy jest trudne. Niskopoziomowe mechanizmy zwiększania niezawodności, np. te z protokołu TCP, działają całkiem dobrze, dlatego błędy na wyższym poziomie zdarzają się rzadko. Bardzo wygodne byłoby ujęcie mechanizmów zapewniania odporności na błędy na wyższych poziomach w abstrakcyjnej postaci, tak by nie trzeba było się tym przejmować w kodzie aplikacji. Obawiam się jednak, że do tej pory nie udało się znaleźć odpowiedniej abstrakcji.

Transakcje od dawna są uznawane za dobrą abstrakcję. Wierzę, że są użyteczne. We wprowadzeniu do rozdziału 7. napisano, że uwzględniają one wiele możliwych problemów (takich jak jednoczesne zapisy, naruszenia więzów, awarie, zakłócenia w sieci, błędy dysków) i sprowadzają je do dwóch

wyników: zatwierdzenia lub anulowania. Jest to wielkim uproszczeniem modelu programowania, obawiam się jednak, że to za mało.

Transakcje są kosztowne — zwłaszcza gdy obejmują niejednorodne technologie składowania danych (zob. punkt „Transakcje rozproszone w praktyce”). Gdy nie chcemy się posługiwać transakcjami rozproszonymi, ponieważ są zbyt kosztowne, musimy sami implementować w kodzie aplikacji mechanizmy zapewniania odporności na błędy. Liczne przykłady z tej książki pokazały, że analizowanie współbieżności i częściowych awarii jest trudne i nieintuicyjne. Dlatego podejrzewam, że większość mechanizmów z poziomu aplikacji nie działa poprawnie. Konsekwencją tego jest utrata lub uszkodzenie danych.

Z tych przyczyn uważam, że warto się zastanowić nad abstrakcjami zapewniającymi odporność na błędy. Takie abstrakcje powinny pozwalać na łatwe zapewnianie poprawności między punktami końcowymi w konkretnych aplikacjach, a jednocześnie zachowywać wysoką wydajność i odpowiednią charakterystykę operacyjną w rozbudowanym środowisku rozproszonym.

## Wymuszanie przestrzegania więzów

Warto się zastanowić nad poprawnością w kontekście pomysłów związanych z podziałem baz na komponenty (zob. punkt „Podział baz na komponenty”). Dowiedziałeś się, że zapobieganie duplikatom na drodze między punktami końcowymi można uzyskać za pomocą identyfikatorów żądań przekazywanych na całej trasie od klienta do bazy rejestrującej zapis. A co z innymi rodzajami więzów?

Tu omówione zostaną przede wszystkim więzy unikatowości — takie jak te, na których polegało na listingu 12.2. W punkcie „Więzy i gwarancje unikatowości” zobaczyłeś kilka innych stosowanych w aplikacjach przykładowych mechanizmów wymagających wymuszania unikatowości. Nazwa lub adres e-mail muszą jednoznacznie identyfikować użytkownika, usługa przechowywania plików nie może obejmować więcej niż jednego pliku o danej nazwie, a dwie osoby nie mogą zarezerwować tego samego miejsca w samolocie lub teatrze.

Inne rodzaje więzów wyglądają bardzo podobnie. Jest tak np. wtedy, gdy chcesz zagwarantować, że stan konta nigdy nie będzie ujemny, że nie sprzedasz więcej sztuk produktu niż masz w magazynie lub że rezerwacje sali konferencyjnej nie będą się pokrywać. Techniki wymuszania unikatowości często można wykorzystać także dla więzów tego rodzaju.

### Więzy unikatowości wymagają konsensusu

W rozdziale 9. zobaczyłeś, że w środowisku rozproszonym wymuszanie więzów unikatowości wymaga konsensusu. Jeśli kilka jednoczesnych żądań dotyczy tej samej wartości, system musi jakoś zdecydować, która ze stojących ze sobą w sprzeczności operacji zostanie zaakceptowana, i odrzucić pozostałe z powodu naruszenia więzów.

Najczęściej stosowany sposób osiągnięcia takiego konsensusu polega na mianowaniu jednego węzła liderem odpowiedzialnym za podejmowanie wszystkich decyzji. To rozwiązanie działa prawidłowo, o ile nie przeszkadza Ci przysyłanie wszystkich żądań do jednego węzła (nawet gdy klient znajduje się na drugim końcu świata), dopóki nie dotknie go awaria. Jeśli musisz zapewnić odporność na awarię lidera, ponownie pojawia się problem osiągnięcia konsensusu (zob. punkt „Replikacja z jednym liderem a konsensus”).

Proces sprawdzania unikatowości jest skalowalny. Wymaga to podziału na partycje na podstawie wartości, która ma być unikatowa. Na przykład jeśli musisz zapewniać unikatowość opartą na identyfikatorach żądań (tak jak na listingu 12.2), możesz sprawić, by wszystkie żądania o tym samym identyfikatorze były kierowane do tej samej partycji (zob. rozdział 6.). Jeżeli chcesz, by unikatowe były nazwy użytkownika, możesz dokonać podziału według skrótów takich nazw.

Jednak wykluczona jest wtedy asynchroniczna replikacja z wieloma węzłami nadrzędnymi. Może się bowiem zdarzyć, że różne węzły nadrzędne jednocześnie zaakceptują sprzeczne zapisy, a tym samym wartości przestaną być unikatowe (zob. punkt „Implementowanie systemów liniowych”). Jeśli chcesz móc natychmiast odrzucać wszystkie zapisy, które mogą naruszyć więzy, nie unikniesz synchronicznej koordynacji [56].

### Unikatowość przy przesyłaniu komunikatów z wykorzystaniem dziennika

Dziennik gwarantuje, że wszyscy użytkownicy zobaczą komunikaty w tej samej kolejności. Jest to gwarancja formalnie nazywana *rozgłaszaniem z uporządkowaniem całkowitym*. To problem analogiczny do osiągnięcia konsensusu (zob. punkt „Rozgłaszanie z uporządkowaniem całkowitym”). W bazach z podziałem na komponenty, które przesyłają komunikaty z wykorzystaniem dziennika, w celu wymuszania więzów unikatowości można zastosować bardzo podobne podejście.

System przetwarzania strumieniowego pobiera wszystkie komunikaty z partycji dziennika sekwencyjnie w jednym wątku (zob. punkt „Porównanie dzienników z tradycyjną obsługą komunikatów”). Dlatego jeśli dziennik jest dzielony według wartości, która ma być unikatowa, system przetwarzania strumieniowego może jednoznacznie i deterministycznie określić, która ze sprzecznych operacji była pierwsza. Gdy kilku użytkowników próbuje zająć tę samą nazwę, wygląda to tak:

1. Każde żądanie nazwy użytkownika jest zapisywane jako komunikat i dołączane do partycji określonej na podstawie skrótu tej nazwy.
2. System przetwarzania strumieniowego sekwencyjnie wczytuje żądania z dziennika, używając lokalnej bazy do śledzenia, które nazwy użytkownika są zajęte. Dla każdego żądania dostępnej nazwy system rejestruje ją jako zajęta i przekazuje komunikat o powodzeniu do strumienia wyjściowego. Dla każdej niedostępnej nazwy system przesyła do strumienia wyjściowego komunikat o odrzuceniu żądania.
3. Klient, który zażądał nazwy, obserwuje strumień wyjściowy i oczekuje na komunikat o powodzeniu lub odrzuceniu żądania.

Ten algorytm jest w zasadzie taki sam jak w punkcie „Implementowanie baz liniowych za pomocą rozgłaszania z uporządkowaniem całkowitym”. Dobrze skaluje się on na potrzeby wysokiej liczby żądań — wystarczy zwiększyć liczbę partycji, tak by każdą z nich przetwarzać niezależnie od pozostałych.

To podejście działa nie tylko dla więzów unikatowości, ale też dla więzów wielu innych rodzajów. Podstawowa zasada polega na tym, że zapisy, które mogą być ze sobą niezgodne, są kierowane do tej samej partycji i przetwarzane sekwencyjnie. W punktach „Czym jest konflikt?” i „Zapis niszczący i fantomy” opisano, że definicja konfliktu zależy od aplikacji. Jednak system przetwarzania strumieniowego może się posłużyć dowolną logiką do sprawdzania poprawności żądań. Ten pomysł jest przypomina podejście zaproponowane po raz pierwszy przez Bayou w latach 90. [58].

## Przetwarzanie żądań dotyczących wielu partycji

Upewnianie się, że operacja jest wykonywana atomowo, a jednocześnie zgodna z więzami, jest ciekawsze, gdy uwzględnia się więcej partycji. Na listingu 12.2 mogą występować trzy partycje: jedna z identyfikatorem żądania, jedna z kontem odbiorcy płatności i jedna z kontem płatnika. Wszystkie te dane są niezależne od siebie, dlatego nie ma powodu, dla którego miałyby się znajdować w tej samej partycji.

W tradycyjnie używanych bazach wykonanie transakcji wymaga atomowego zatwierdzenia operacji we wszystkich trzech partycjach. W praktyce zmusza to do zapewnienia uporządkowania całkowitego wszystkich innych transakcji z każdej z tych partycji. Ponieważ koordynacja dotyczy różnych partycji, nie można ich przetwarzać niezależnie, dlatego przepustowość systemu zapewne spadnie.

Okazuje się jednak, że poprawność można też uzyskać dzięki podzielonym na partycje dziennikom i bez zatwierdzania atomowego:

1. Żądanie przelewu środków z konta A na konto B otrzymuje unikatowy identyfikator od klienta i jest dołączane do partycji dziennika na podstawie tego identyfikatora.
2. System przetwarzania strumieniowego wczytuje dziennik żądań. Dla każdego komunikatu z żądaniem do strumienia wyjściowego przesyłane są dwa komunikaty: polecenie obciążenia płatnika (konta A) i polecenie uznania odbiorcy płatności (konta B). Te komunikaty są kierowane do partycji właściwych dla tych kont. W przekazywanych komunikatach umieszczany jest identyfikator pierwotnego żądania.
3. Dalsze systemy pobierają strumień poleceń obciążenia i uznania konta, deduplikują je na postawie identyfikatorów żądania i zmieniają stan kont.

Kroki 1. i 2. są niezbędne, ponieważ jeśli klient bezpośrednio prześle polecenia obciążenia i uznania, konieczne jest atomowe zatwierdzanie operacji w dwóch partycjach, aby mieć pewność, że albo wykonane zostaną oba zadania, albo żadne z nich. Żeby uniknąć transakcji rozproszonej, najpierw żądanie jest trwale zapisywane jako jeden komunikat. Następnie na jego podstawie generowane są polecenia obciążenia i uznania. Zapisy jednego obiektu w prawie wszystkich systemach danych są atomowe (zob. punkt „Zapisy pojedynczego obiektu”). Dlatego żądanie albo pojawia się w dzienniku, albo nie. Nie jest konieczne atomowe zatwierdzanie w ramach wielu partycji.

Jeśli system przetwarzania strumieniowego z etapu 2. ulegnie awarii, wznowi pracę od najnowszego punktu kontrolnego. Nie pomija przy tym żadnych komunikatów z żądaniami, przy czym może przetwarzać je kilkakrotnie i generować zduplikowane polecenia obciążenia i uznania. Jednak ponieważ działa deterministycznie, ponownie utworzy te same polecenia, a systemy z kroku 3. mogą łatwo zdeduplikować je za pomocą identyfikatora żądania używanego między punktami końcowymi.

Jeśli chcesz mieć pewność, że stan konta płatnika nie stanie się ujemny, możesz zastosować dodatkowy system przetwarzania strumieniowego (dzielony na partycje według numerów kont płatników), który aktualizuje stan kont i sprawdza poprawność transakcji. Wtedy w dzienniku żądań w kroku 1. umieszczane byłyby tylko poprawne transakcje.

Podział transakcji z udziałem wielu partycji na dwa etapy z różnym podziałem na partycje i zastosowanie identyfikatora żądania na całej drodze między punktami końcowymi pozwala uzyskać te same co wcześniej gwarancje poprawności (każde żądanie jest uwzględniane tylko raz zarówno na koncie płatnika, jak i na koncie odbiorcy) nawet w obliczu błędów i to bez stosowania protokołu zatwierdzania atomowego. Pomysł wykorzystania kilku etapów z różnym podziałem na partycje jest podobny do rozwiązania opisanego w punkcie „Przetwarzanie danych w wielu partycjach” (zob. też punkt „Kontrola współbieżności”).

## Aktualność a integralność

Wygodną cechą transakcji jest to, że zwykle są liniowe (zob. punkt „Liniowość”). Oznacza to, że jednostka zapisująca zwykle czeka na zatwierdzenie transakcji, po czym zapisy stają się natychmiast widoczne dla wszystkich jednostek odczytujących.

Sytuacja wygląda inaczej, gdy operacja jest rozbijana na kilka etapów w systemach przetwarzania strumieniowego. Konsumenci dziennika z natury działają asynchronicznie, dlatego nadawca nie czeka na przetworzenie przez nich jego komunikatu. Klient może jednak oczekiwać na pojawienie się komunikatu w strumieniu wyjściowym. To podejście zastosowano w punkcie „Unikatowość przy przesyłaniu komunikatów z wykorzystaniem dziennika” do sprawdzania, czy więzy unikatowości są przestrzegane.

W przedstawionym przykładzie poprawność sprawdzania unikatowości nie zależy od tego, czy nadawca komunikatu czeka na wynik. Celem oczekiwania jest tylko synchroniczne informowanie nadawcy o tym, czy sprawdzanie unikatowości zakończyło się powodzeniem. Jednak powiadomienia można oddzielić od efektów przetwarzania komunikatu.

Na ogólnym poziomie uważam, że pojęcie *spójność* łączy dwa różne wymagania, które warto rozważać osobno:

### *Aktualność*

Aktualność oznacza gwarantowanie, że użytkownicy zobaczą system w aktualnym stanie. Wcześniej dowiedziałeś się, że jeśli użytkownik wczytuje dane z przestarzałej kopii, może je uzyskać w niespójnym stanie (zob. punkt „Problemy z opóźnieniem replikacji”). Jednak taka niespójność jest tymczasowa i ostatecznie zostanie wyeliminowana dzięki oczekiwaniu i ponowieniu próby.

W twierdzeniu CAP (zob. punkt „Koszty liniowości”) pojęcia „spójność” używa się w sensie liniowości, która jest wymagającym sposobem zapewniania spójności. Przydatne mogą być też mniej wymagające cechy, które wiążą się z aktualnością, np. *odczyt po zapisie* (zob. punkt „Odczyty własnych zapisów”).

### *Integralność*

Integralność oznacza brak uszkodzeń (nie nastąpiła utrata danych oraz nie są one sprzeczne ze sobą lub nieprawdziwe). Przede wszystkim jeśli jakiś pochodny zbiór danych jest aktualizowany jako widok podstawowych danych (zob. punkt „Generowanie aktualnego stanu na podstawie dziennika zdarzeń”), proces generowania go musi być prawidłowy. Na przykład indeks bazy musi poprawnie odzwierciedlać jej zawartość. Indeks, w którym brakuje niektórych rekordów, nie jest specjalnie przydatny.

Po naruszeniu integralności niespójność staje się trwała. Oczekiwanie i ponowienie próby zwykle nie naprawi uszkodzenia bazy. Zamiast tego trzeba bezpośrednio sprawdzić i poprawić dane. W kontekście transakcji ACID (zob. punkt „Znaczenie gwarancji ACID”) spójność jest zwykle traktowana jak specyficzne dla aplikacji ujęcie integralności. Ważnymi narzędziami zapewniania integralności są atomowość i trwałość.

W formie hasła można to ująć tak: naruszenie aktualności to „spójność ostateczna”, a naruszenie integralności to „wieczna niespójność”. Moim zdaniem w większości aplikacji integralność jest znacznie ważniejsza od aktualności. Naruszenie aktualności może być irytujące i dezorientujące, jednak naruszenie integralności może prowadzić do katastrofy.

Nie jest zaskoczeniem np. to, że w bilansie karty kredytowej nie widać jeszcze transakcji z ostatnich 24 godzin. To normalne, że systemy mają określone opóźnienie. Wiadomo, że banki uzgadniają transakcje asynchronicznie, a aktualność nie ma tu kluczowego znaczenia [3]. Jest jednak poważnym problemem, gdy bilans nie równa się sumie transakcji plus wcześniejszy stan (błąd w dodawaniu) lub gdy zostajesz obciążony, ale środki nie trafiają do sprzedawcy (zniknięcie środków). Takie kłopoty oznaczają naruszenie integralności systemu.

### **Poprawność systemów przepływu danych**

Transakcje ACID zwykle gwarantują zarówno aktualność (np. liniowość) i integralność (np. zatwierdzanie atomowe). Dlatego jeśli patrzysz na poprawność aplikacji z perspektywy transakcji ACID, różnica między aktualnością a integralnością nie ma znaczenia.

Jednak ciekawą cechą omawianych w tym rozdziale systemów przepływu danych opartych na zdarzeniach jest to, że oddzielają aktualność od integralności. W trakcie asynchronicznego przetwarzania strumieni zdarzeń aktualność nie jest gwarantowana, chyba że celowo zbudujesz konsumentów oczekujących na nadejście komunikatu i dopiero potem zwracających sterowanie. W systemach przetwarzania strumieniowego najważniejsza jest integralność.

Semantyka *dokładnie jednokrotnego* lub *praktycznie jednokrotnego* przetwarzania (zob. punkt „Odporność na błędy”) jest mechanizmem zapewniania integralności. Jeśli zdarzenie zostanie zgubione lub jego efekty wystąpią dwukrotnie, integralność systemu może zostać naruszona. Dlatego dostarczanie komunikatów z zachowaniem odporności na błędy i zapobieganie duplikatom (np. za pomocą operacji idempotentnych) jest ważne, aby zapewnić integralność systemu danych w obliczu usterek.

W poprzednim punkcie pokazano, że niezawodne systemy przetwarzania strumieniowego mogą zachować integralność bez transakcji rozproszonych i protokołu zatwierdzania atomowego. To oznacza, że mogą zapewnić porównywalną poprawność przy znacznie wyższej wydajności i stabilności operacyjnej. Integralność jest tu osiągnięta dzięki połączeniu kilku mechanizmów:

- Reprezentowanie zawartości operacji zapisu jako jednego komunikatu, który można zarejestrować atomowo. To podejście świetnie pasuje do techniki event sourcing (zob. punkt „Event sourcing”).
- Generowanie wszystkich pozostałych aktualizacji stanu na podstawie tego jednego komunikatu z użyciem deterministycznych funkcji generujących (podobnie działają procedury składowane; zob. punkty „Rzeczywiste sekwencyjne wykonywanie transakcji” i „Kod aplikacji jako funkcja generująca”).

- Przekazywanie wygenerowanego przez klienta identyfikatora żądania na wszystkich poziomach przetwarzania, co umożliwia zapobieganie duplikatom i idempotencję na drodze od jednego punktu końcowego do drugiego.
- Niemodyfikowalność komunikatów i umożliwianie okresowego ponownego przetwarzania danych pochodnych, co ułatwia przywracanie stanu po błędach (zob. punkt „Zalety niemodyfikowalnych zdarzeń”).

To połączenie mechanizmów wydaje mi się bardzo obiecującym kierunkiem budowania w przyszłości aplikacji odpornych na błędy.

## Luźno interpretowane więzy

Wcześniej opisano, że wymuszanie więzów unikatowości wymaga konsensusu — zwykle zapewnianego przez przekazywanie wszystkich zdarzeń z konkretnej partycji do jednego węzła. To ograniczenie jest nieuniknione, jeśli potrzebujesz więzów unikatowości w ich tradycyjnej postaci. W przetwarzaniu strumieniowym nie da się tego uniknąć.

Należy też zwrócić uwagę na to, że w wielu rzeczywistych aplikacjach wystarczają znacznie słabsze gwarancje unikatowości:

- Jeśli dwie osoby jednocześnie rejestrują tę samą nazwę lub rezerwują to samo miejsce, można przesłać jednej z nich komunikat z przeprosinami i poprosić o zmianę wyboru. Tego rodzaju zmiana korygująca błąd to *transakcja kompensująca* [59, 60].
- Jeżeli konsument zamawia więcej produktów, niż masz w magazynie, możesz ściągnąć dodatkowe sztuki, przeprosić klienta za opóźnienie i zaoferować mu rabat. Sytuacja jest wtedy taka sama, jakby np. wózek widłowy zmiażdżył część produktów w magazynie, przez co liczba sztuk jest mniejsza niż oczekiwana [61]. Dlatego przepływ pracy z przeprosinami i tak powinien być jednym z procesów biznesowych. Nie zawsze więc są konieczne liniowe ograniczenia dotyczące liczby produktów w magazynie.
- Podobnie wiele linii lotniczych sprzedaje więcej biletów, niż jest miejsc, oczekując, że niektórzy pasażerowie spóźnią się na lot. Także wiele hoteli przyjmuje rezerwacje większej liczby pokoi, zakładając, że niektórzy goście anulują przyjazd. W takich sytuacjach ograniczenie „jedna osoba na miejsce” jest celowo naruszane z powodów biznesowych, a procesy prowadzące do rekompensaty (zwrotu pieniędzy, zaoferowania wyższego standardu, zapewnienia zastępczego pokoju w pobliskim hotelu) są opracowywane na potrzeby radzenia sobie z przypadkami, w których popyt przewyższa podaż. Nawet bez przyjmowania zbyt wielu rezerwacji procesy obejmujące przeprosiny i rekompensatę są potrzebne, aby radzić sobie z anulowaniem lotów z powodu złej pogody lub strajku. Powrót do normy po takich problemach to normalny aspekt funkcjonowania firmy [3].
- Jeśli ktoś wypłaca więcej pieniędzy niż ma na koncie, bank może obciążyć go kosztami debetu i zażądać zwrotu długu. Dzięki limitowi kwoty wypłat w ciągu dnia ryzyko banku jest ograniczone.

W wielu kontekstach biznesowych akceptowalne jest tymczasowe naruszenie więzów i późniejsza naprawa sytuacji za pomocą przeprosin. Koszt przeprosin (mierzony pieniędzmi lub reputacją) bywa różny, jednak często jest stosunkowo niski. Nie da się wycofać przesłanego e-maila, ale można

później wysłać wiadomość korygującą. Jeśli przypadkowo dwukrotnie obciążysz kartę kredytową klienta, możesz zwrócić kwotę z jednej z tych operacji. Twoje koszty to wtedy tylko opłaty transakcyjne i ewentualne skargi klienta. Po wypłaceniu środków z bankomatu nie można bezpośrednio ich odzyskać, choć teoretycznie można wysłać windykatorów, aby odzyskali pieniądze, jeśli stan konta został przekroczony i klient nie chce zwrócić długu.

Stwierdzenie, czy koszty przeprosin są akceptowalne, to decyzja biznesowa. Jeśli są one do przyjęcia, tradycyjny model sprawdzania wszystkich więzów przed zapisem danych jest niepotrzebnie restrykcyjny, a więzy związane z liniowością są niepotrzebne. Sensownym rozwiązaniem może być zapis optymistyczny i sprawdzanie więzów po fakcie. Nadal możesz przeprowadzać walidację przed wykonaniem operacji, których wycofanie byłoby kosztowne. Nie oznacza to jednak, że musisz sprawdzać poprawność jeszcze przed zapisem danych.

Aplikacje *wymagają* integralności. Nie chcesz gubić rezerwacji lub pieniędzy z powodu niedopasowania obciążeń i uznań. Jednak wymuszanie więzów *nie* musi przebiegać od razu. Jeśli sprzedałeś więcej produktów, niż masz ich w magazynie, możesz naprawić problem po fakcie, przepraszając. Jest to podobne do technik rozwiązywania konfliktów, opisanych w punkcie „Radzenie sobie z konfliktami przy zapisie”.

## Systemy danych unikające koordynacji

Poczynione zostały dwie ciekawe obserwacje:

1. Systemy przepływu danych mogą zapewniać gwarancje integralności danych pochodnych bez zatwierdzania atomowego, liniowości i synchronicznej koordynacji między partycjami.
2. Choć ściśle więzy unikatowości wymagają aktualności i koordynacji, w wielu aplikacjach dopuszczalne są luźniejsze więzy, które mogą zostać tymczasowo zerwane i później naprawione (o ile integralność zostaje zachowana).

Te dwie obserwacje oznaczają, że systemy przepływu danych mogą zapewniać usługi zarządzania danymi w wielu aplikacjach bez wymogu koordynacji, a przy tym wciąż oferować mocne gwarancje integralności. Takie systemy danych *unikające koordynacji* są bardzo atrakcyjne, ponieważ pozwalają uzyskać wyższą wydajność i odporność na błędy niż systemy wymagające synchronicznej koordynacji [56].

Ten system mógłby działać (jako rozproszony) w wielu centrach danych w konfiguracji z wieloma liderami i asynchroniczną replikacją między regionami. Każde centrum mogłoby wtedy kontynuować pracę niezależnie od pozostałych, ponieważ nie jest konieczna synchroniczna koordynacja między regionami. System oferowałby słabe gwarancje aktualności (nie mógłby być liniowy bez wprowadzenia koordynacji), ale silne gwarancje integralności.

W tym kontekście transakcje sekwencyjne nadal są przydatne do aktualizowania stanu pochodnego, jednak można je stosować w mniejszym zakresie — tam, gdzie dobrze działają [8]. Niejednorodne transakcje rozproszone, np. transakcje XA (zob. punkt „Transakcje rozproszone w praktyce”), nie są konieczne. Koordynację synchroniczną można wprowadzić tam, gdzie to niezbędne (np. w celu wymuszania przestrzegania ścisłych więzów w operacjach, po których nie da się przywrócić stanu). Nie trzeba jednak wszędzie ponosić kosztów koordynacji, jeśli jest ona potrzebna tylko w niewielkiej części aplikacji [43].



Można też spojrzeć na koordynację i więzy w inny sposób — ograniczają one liczbę przeprosin, jakie są potrzebne z powodu niespójności, ale mogą jednocześnie pogarszać wydajność i dostępność systemu, a tym samym zwiększać liczbę przeprosin koniecznych z przyczyny przestojów. Nie da się całkiem wyeliminować przeprosin, można jednak dążyć do osiągnięcia możliwie najlepszego kompromisu — złotego środka, w którym nie występuje ani za dużo niespójności, ani za dużo problemów z dostępnością.

## Ufaj, ale kontroluj

Wszystkie opisy poprawności, integralności i odporności na błędy były oparte na założeniu, że niektóre rzeczy mogą pójść źle, ale w innych miejscach problemy nie wystąpią. Te założenia to część *modelu systemu* (zob. punkt „Odzworowywanie modeli systemu na świat rzeczywisty”). Należy np. przyjąć, że procesy mogą ulec awarii, maszyny mogą nagle utracić zasilanie, a sieć może opóźniać lub gubić komunikaty. Jednocześnie można założyć, że po wywołaniu fsync dane na dysku nie zostaną utracone, że dane w pamięci nie są uszkodzone i że instrukcja mnożenia w procesorze zawsze zwraca prawidłowy wynik.

Te założenia są całkiem rozsądne i przeważnie prawdziwe. Trudno byłoby cokolwiek zrobić, gdyby trzeba było cały czas się martwić o to, że komputery się pomylą. Tradycyjnie w modelach systemów przyjmuje się binarne podejście do błędów. Zakładamy, że niektóre rzeczy mogą się wydarzyć, a inne nigdy się nie dzieją. W praktyce ważniejsze jest prawdopodobieństwo. Niektóre sytuacje są bardziej prawdopodobne, inne mniej. Ważne jest, czy naruszenie założeń zdarza się na tyle często, że możemy natknąć się na nie w praktyce.

Zobaczyłeś już, że dane mogą zostać uszkodzone, gdy nieużywane znajdują się na dysku (zob. ramka „Replikacja i trwałość”). Ponadto uszkodzenie danych w sieci może zostać niewykryte za pomocą sum kontrolnych z protokołu TCP (zob. punkt „Słabe formy kłamania”). Może jest to kwestia, na którą warto zwrócić więcej uwagi?

Pewna aplikacja, nad którą kiedyś pracowałem, zbierała raporty o awariach od klientów. Niektóre ze zgłoszonych usterek można było wytłumaczyć wyłącznie losową zmianą wartości bitów w pamięci urządzeń. Wydaje się to nieprawdopodobne, ale jeśli oprogramowanie działa w wystarczająco wielu urządzeniach, wydarzyć się mogą nawet bardzo zaskakujące rzeczy. Obok losowego uszkodzenia pamięci z powodu błędów sprzętowych lub promieniowania niektóre „patologiczne” wzorce dostępu do pamięci mogą zmieniać wartość bitów nawet w sytuacji, gdy błędy nie występują [62]. Ten efekt można wykorzystać do łamania zabezpieczeń w systemach operacyjnych [63] (ta technika to *rowhammer*). Gdy z bliska przyjrzyysz się sprzętowi, zobaczysz, że nie jest on tak idealną abstrakcją, jak się może wydawać.

Warto zauważyć, że losowe zmiany wartości bitów w nowoczesnym sprzęcie są bardzo rzadkie [64]. Chciałem jedynie pokazać, że nie są one nieprawdopodobne, dlatego zasługują na uwagę.

## Zachowywanie integralności w obliczu błędów programowych

Obok tego rodzaju błędów sprzętowych zawsze istnieje ryzyko wystąpienia błędów programowych, które nie są wykrywane za pomocą niskopoziomowych sum kontrolnych w sieci, pamięci lub systemie plików. Nawet powszechnie stosowane oprogramowanie bazodanowe zawiera błędy. Sam

widziałem, jak baza MySQL nie zachowywała prawidłowo więzów unikatowości [65], a na poziomie izolacji sekwencyjnej w bazie PostgreSQL występuje anomalia zapisu zniekształcającego [66]. A MySQL i PostgreSQL to stabilne i cenione bazy, przetestowane w praktyce przez liczne osoby przez wiele lat. W mniej dojrzałym oprogramowaniu sytuacja jest zapewne dużo gorsza.

Mimo znacznych wysiłków włożonych w staranne projektowanie, testowanie i recenzowanie oprogramowania błędy i tak się pojawiają. Choć są rzadkie i ostatecznie zostają znalezione oraz naprawione, przez pewien okres mogą uszkadzać dane.

Jeśli chodzi o kod aplikacji, trzeba założyć, że błędów jest w nim o wiele więcej. Wynika to z tego, że większość aplikacji nie jest recenzowana i testowana tak dokładnie jak bazy danych. W wielu aplikacjach nie są nawet poprawnie stosowane funkcje używane w bazach do zapewniania integralności (np. klucze obce lub więzy unikatowości) [36].

Spójność (w sensie używanym w modelu ACID; zob. punkt „Spójność”) jest oparta na założeniu, że baza rozpoczyna pracę w spójnym stanie, a transakcja przekształca ją z jednego spójnego stanu w inny, też spójny. Dlatego oczekujemy, że baza zawsze jest spójna. Ma to jednak sens tylko wtedy, gdy transakcja jest wolna od błędów. Jeśli aplikacja niepoprawnie używa bazy, np. w niebezpieczny sposób korzysta z niskich poziomów izolacji, nie można zagwarantować integralności bazy.

### Nie wierz bezkrytycznie temu, co obiecują producenci

Ponieważ zarówno sprzęt, jak i oprogramowanie nie zawsze jest tak idealne, jak byśmy chcieli, wydaje się, że uszkodzenie danych jest nieuniknione. Dlatego należy przynajmniej znaleźć sposób ustalenia, czy w danych pojawiły się błędy, aby można je było naprawić i spróbować dotrzeć do źródła problemu. Sprawdzanie integralności danych to *audytowanie*.

W punkcie „Zalety niemodyfikowalnych zdarzeń” opisano, że audytowanie dotyczy nie tylko aplikacji finansowych. Jednak w świecie finansów jest ono bardzo ważne, ponieważ każdy wie, że pomyłki się zdarzają, i akceptuje konieczność wykrywania oraz naprawiania problemów.

W dojrzałych systemach też zwykle uwzględnia się możliwość wystąpienia mało prawdopodobnych trudności i stosuje się odpowiednie zabezpieczenia. Na przykład twórcy działających w dużej skali systemów składowania danych (takich jak HDFS i Amazon S3) nie ufają w pełni dyskom, dlatego uruchamiają działające w tle procesy, które nieustannie wczytują pliki, porównują je ze starszymi replikami i przenoszą pliki między dyskami w celu ograniczenia ryzyka niezauważalnego uszkodzenia danych [67].

Jeśli chcesz mieć pewność, że dane wciąż są dostępne, musisz je wczytać i sprawdzić. Zwykle dane są na miejscu, jeżeli jednak dzieje się inaczej, lepiej dowiedzieć się o tym wcześniej niż później. Z tych samych przyczyn ważne jest, aby od czasu do czasu spróbować odzyskać dane z kopii zapasowej. W przeciwnym razie możesz się dowiedzieć o uszkodzeniach kopii, gdy jest za późno i już utraciłeś dane. Nie zakładaj bezkrytycznie, że wszystko działa poprawnie!

### Kultura weryfikowania

W systemach takich jak HDFS i S3 trzeba przyjmować, że dyski przez większość czasu działają prawidłowo. Jest to sensowne założenie, które się różni od przyjmowania, że *zawsze* funkcjonują one poprawnie. Jednak w niewielu systemach stosuje się obecnie podejście „ufaj, ale kontroluj”

i ciągle audyty swojego stanu. W licznych systemach przyjmuje się, że gwarancje poprawności są bezwzględne, i nie bierze się pod uwagę możliwości wystąpienia rzadkich uszkodzeń danych. Mam nadzieję, że w przyszłości pojawi się więcej systemów *samosprawdzających się* lub *samoaudytujących*, które nieustannie będą kontrolować swoją integralność, zamiast polegać na ślepych zaufaniu [68].

Obawiam się, że kultura baz ACID doprowadziła do budowania aplikacji z bezkrytycznym zaufaniem do technologii (np. transakcji) i z zaniedbywaniem jakichkolwiek audytów. Ponieważ technologie, którym ufaliśmy, zwykle działały wystarczająco dobrze, mechanizmy potrzebne do audytów wydawały się niewarte inwestowania w nie.

Jednak później środowisko baz danych się zmieniło. W świecie baz NoSQL słabsze gwarancje spójności stały się normą, a mniej dojrzałe technologie składowania danych trafiły do powszechnego użytku. Ale ponieważ nie opracowano mechanizmów prowadzenia audytów, nadal rozwijamy aplikacje na podstawie bezkrytycznego zaufania, choć obecnie to podejście stało się bardziej niebezpieczne. Warto zastanowić się przez chwilę nad projektowaniem z myślą o możliwości prowadzenia audytów.

### Projektowanie pod kątem możliwości audytowania

Jeśli transakcja modyfikuje kilka obiektów w bazie, trudno jest później stwierdzić, co ta transakcja oznacza. Nawet jeżeli sprawdzisz dzienniki transakcji (zob. punkt „Przechwytywanie zmian w danych”), operacje wstawiania, aktualizowania i usuwania danych w różnych tabelach niekoniecznie dadzą Ci jasny obraz tego, *dlaczego* wprowadzono zmiany. Wywołanie logiki aplikacji, która zdecydowała o tych modyfikacjach, jest przejściowe i nie da się go odtworzyć.

Z kolei systemy oparte na zdarzeniach mogą ułatwiać audytowanie. W technice event sourcing przesyłane do systemu dane wejściowe od użytkownika są reprezentowane jako jedno niemodyfikowalne zdarzenie. Wynikowe aktualizacje stanu są generowane na podstawie tego zdarzenia. Proces generowania może być deterministyczny i powtarzalny. Dlatego przekazanie tego samego dziennika zdarzeń do tej samej wersji kodu generującego da w efekcie te same aktualizacje stanu.

Bezpośrednie określenie przepływu danych (zob. punkt „Filozofia obsługi danych wyjściowych z procesów wsadowych”) pozwala dużo łatwiej ustalić ich *pochodzenie*. Dzięki temu sprawdzanie integralności staje się znacznie prostsze. W przypadku dziennika zdarzeń można zastosować skróty do sprawdzania, czy magazyn zdarzeń nie został uszkodzony. Jeśli chodzi o stan pochodny, można ponownie uruchomić systemy przetwarzania wsadowego i strumieniowego, które wygenerowały stan na podstawie dziennika zdarzeń, aby sprawdzić, czy wynik jest taki sam. Można nawet równolegle wykonywać nadmiarowy proces generowania.

Deterministyczny i dobrze zdefiniowany przepływ danych ułatwia też debugowanie i śledzenie pracy systemu w celu ustalenia, *dlaczego* coś zrobił [4, 69]. Jeśli nastąpiło coś nieoczekiwanego, warto mieć narzędzia diagnostyczne pozwalające odtworzyć okoliczności, które doprowadziły do niespodziewanego zdarzenia. Jest to swoiste narzędzie do podróży w czasie w trakcie debugowania.

### Jeszcze o zasadzie punktów końcowych

Skoro nie można w pełni ufać temu, że wszystkie komponenty systemu będą wolne od błędów (czyli że każde urządzenie i każdy program są pozbawione usterek), trzeba przynajmniej okresowo sprawdzać integralność danych. W przeciwnym razie dowiesz się o uszkodzeniu, gdy będzie już

za późno i gdy wystąpiły problemy na dalszych etapach przetwarzania. Wtedy ustalanie źródła problemu jest znacznie trudniejsze i bardziej kosztowne.

Sprawdzanie integralności systemów danych najlepiej przeprowadzać na całej trasie między punktami końcowymi (zob. punkt „Zasada punktów końcowych dla baz danych”). Im więcej systemów uwzględniysz w trakcie sprawdzania integralności, tym mniejsze wystąpi zagrożenie, że na jakimś etapie procesu uszkodzenie danych pozostanie niezauważone. Jeśli możesz sprawdzić, że cały potok danych pochodnych między punktami końcowymi jest prawidłowy, uwzględnione zostaną wszystkie dyski, sieci, usługi i algorytmy ze ścieżki.

Ciągłe testy integralności między punktami końcowymi zwiększają pewność co do poprawności systemu, co pozwala na szybszą pracę [70]. Audyty (podobnie jak zautomatyzowane testy) zwiększają prawdopodobieństwo szybkiego znalezienia błędów, a tym samym zmniejszają ryzyko, że zmiana w systemie lub nowa technologia składowania danych spowoduje uszkodzenia. Jeśli nie musisz się obawiać wprowadzania zmian, możesz znacznie sprawniej modyfikować aplikację, aby spełnić nowe wymagania.

### Narzędzia do tworzenia systemów danych umożliwiających audyty

Obecnie w tylko nielicznych systemach danych możliwość audytowania jest istotna. W niektórych aplikacjach implementowane są mechanizmy audytowania (np. wszystkie zmiany rejestruje się w odrębnej tabeli), jednak trudno jest zagwarantować integralność dziennika audytu i stanu bazy. Dziennik transakcji można zabezpieczyć przed manipulacjami, okresowo podpisując go z użyciem sprzętowego modułu zabezpieczeń. To nie gwarantuje jednak, że w dzienniku znalazły się prawidłowe transakcje.

Ciekawe byłoby zastosowanie narzędzi kryptograficznych do dowodzenia integralności systemu w sposób odporny na różnorodne problemy sprzętowe i programowe, a nawet potencjalne złośliwe działania. W ramach eksploracji tego obszaru pojawiły się kryptowaluty, łańcuchy bloków i rozproszone rejestry takie jak Bitcoin, Ethereum, Ripple, Stellar i wiele innych [71, 72, 73].

Nie mam kwalifikacji, aby się wypowiadać na temat zalet tych technologii w obszarach walut lub mechanizmów uzgadniania kontraktów. Jednak w kontekście systemów danych te rozwiązania obejmują ciekawe pomysły. Są one jak rozproszone bazy danych z modelem danych i mechanizmem obsługi transakcji, gdzie różne repliki mogą być utrzymywane przez nieufające sobie organizacje. Te repliki stale sprawdzają integralność pozostałych replik i korzystają z protokołu osiągania konsensusu do uzgadniania, które transakcje należy wykonać.

Nieco sceptycznie podchodzę do aspektów związanych z odpornością tych technologii na błędy bizantyjskie (zob. punkt „Błędy bizantyjskie”). Ponadto uważam technikę generowania *dowodów pracy* (czyli kopania bitcoinów) za niezwykle marnotrawstwo. Przepustowość transakcji w systemie bitcoina jest dość niska, choć wynika to bardziej z powodów politycznych i ekonomicznych niż technicznych. Ciekawe są jednak aspekty sprawdzania integralności.

Kryptograficzne audyty i sprawdzanie integralności często są oparte na *drzewach skrótów* (ang. *Merkle trees*) [74], które można wykorzystać do wydajnego dowodzenia, że rekord występuje w określonym zbiorze danych (oraz kilku innych rzeczy). Poza głośnymi obecnie kryptowalutami drzewa skrótów są wykorzystywane także w dziedzinie *przejrzystości certyfikatów*, gdzie służą do sprawdzania poprawności certyfikatów TLS i SSL [75, 76].

Potrafię sobie wyobrazić, że algorytmy sprawdzania integralności i audytowania (takie jak z obszarów przejrzystości certyfikatów i rozproszonych rejestrów) będą częściej używane w systemach danych. Trzeba włożyć trochę pracy w to, aby uzyskać skalowalność podobną jak w systemach bez audytów kryptograficznych i ograniczyć spadek wydajności. Myślę jednak, że jest to ciekawy obszar, który w przyszłości warto obserwować.

## Robienie tego, co słuszne

W ostatnim fragmencie tej książki chcę przedstawić szerszą perspektywę. W tej publikacji omówiono szeroki zestaw architektur systemów danych, przeanalizowano ich wady i zalety oraz zbadano techniki budowania niezawodnych, skalowalnych i łatwych w konserwacji aplikacji. Pominięto jednak ważną i fundamentalną kwestię, którą chciałbym teraz uzupełnić.

Każdy system jest budowany w jakimś celu. Wszystkie działania, jakie podejmujemy, mają zamierzone i niezamierzone skutki. Cel może być tak prosty jak zarabianie pieniędzy, jednak wpływ systemu na świat może znacznie wykraczać poza pierwotne zamierzenia. My wszyscy, inżynierowie budujący te systemy, odpowiadamy za staranne rozważenie tego wpływu i świadome decydowanie o tym, w jakim świecie chcemy żyć.

Piszę o danych jak o czymś abstrakcyjnym. Warto jednak pamiętać, że wiele zbiorów danych dotyczy ludzi: ich zachowań, zainteresowań, tożsamości. Takie dane trzeba traktować po ludzku i z szacunkiem. Użytkownicy też są ludźmi, a szacunek dla człowieka ma ogromne znaczenie.

Rozwój oprogramowania coraz częściej wiąże się z ważnymi wyborami etycznymi. Dostępne są wytyczne pomagające inżynierom oprogramowania poruszać się w tym obszarze, np. kodeks Software Engineering Code of Ethics and Professional Practice stowarzyszenia ACM [77], jednak rzadko dyskutuje się na ich temat, stosuje je i przestrzega ich w praktyce. W efekcie inżynierowie i menedżerowie produktu czasem bardzo niewiele uwagi poświęcają prywatności i potencjalnie negatywnym skutkom działania ich produktów [78, 79, 80].

Technologia sama w sobie nie jest ani dobra, ani zła. Liczy się to, jak jest używana i jak wpływa na ludzi. Dotyczy to w równym stopniu systemów takich jak wyszukiwarka jak broni takiej jak pistolet. Uważam, że nie wystarczy, aby inżynierowie oprogramowania skupiali się wyłącznie na technologii i ignorowali skutki jej działania. Musimy uwzględniać także kwestie etyczne. Analizowanie etyki jest trudne, jest to jednak zbyt ważny temat, aby go ignorować.

## Analityka prognostyczna

Na przykład analityka prognostyczna jest ważnym aspektem modnego nurtu Big Data. Wykorzystywanie analizy danych do prognozowania pogody lub rozprzestrzeniania się chorób to jedna sprawa [81]. Czymś innym jest szacowanie, czy skazaniec ponownie popełni przestępstwo, czy osoba wnioskująca o kredyt nie będzie go spłacać lub czy ubezpieczony będzie zgłaszał kosztowne roszczenia. W tych sytuacjach analizy mają bezpośredni wpływ na życie konkretnych ludzi.

Naturalne jest, że sieci obsługi płatności chcą unikać fałszywych transakcji, banki wystrzegają się złych długów, linie lotnicze nie chcą porwać, a firmy wzbraniają się przed zatrudnianiem niewydajnych lub niegodnych zaufania pracowników. Z perspektywy tych firm wartość utraty korzyści

biznesowych jest niewielka w porównaniu z kosztem złej pożyczki lub kłopotliwego pracownika. Dlatego zrozumiałe jest, że organizacje postępują ostrożnie. Jeśli mają wątpliwości, wolą odmówić.

Jednak wraz z upowszechnianiem się algorytmicznego podejmowania decyzji osoba, która (właściwie lub nie) została uznana za ryzykowną przez jeden algorytm, będzie narażona na dużą liczbę odmów. Regularne odmowy zatrudnienia, biletów lotniczych, ubezpieczenia zdrowotnego, ubezpieczenia mieszkania, usług finansowych i innych ważnych aspektów życia społecznego są tak poważnym ograniczeniem swobód człowieka, że zostały nazwane „algorytmicznym więzieniem” [82]. W państwach, w których przestrzegane są prawa człowieka, system sprawiedliwości zakłada niewinność podejrzanego do czasu udowodnienia mu winy. Jednak systemy zautomatyzowane mogą regularnie i arbitralnie wykluczać ludzi z uczestnictwa w życiu społecznym bez żadnego dowodu winy i z niewielką szansą na udaną apelację.

## **Upředzenia i dyskryminacja**

Decyzje algorytmów nie są z natury ani lepsze, ani gorsze od tych podejmowanych przez ludzi. Każdy człowiek zwykle ma upředzenia (nawet jeśli aktywnie próbuje z nimi walczyć), a praktyki dyskryminacyjne mogą się stać kulturowo zinstytucjonalizowane. Jest nadzieja, że decyzje podejmowane na podstawie danych (a nie według subiektywnych i instynktownych opinii ludzi) mogą okazać się bardziej uczciwe i dawać więcej szans ludziom często pomijanym w tradycyjnym systemie [83].

Budowanie systemów analityki prognostycznej nie polega tylko na automatyzowaniu podejmowanych przez ludzi decyzji za pomocą oprogramowania, w którym określono reguły dotyczące akceptacji lub odmowy. Same reguły też są ustalane na podstawie danych. Jednak wzorce wykrywane przez te systemy bywają niejasne. Nawet jeśli w danych występuje jakaś korelacja, możemy nie wiedzieć, z czego ona wynika. Jeżeli w danych wejściowych występuje błąd systematyczny, system zapewne zauważy go i dodatkowo wzmocni w danych wyjściowych [84].

W wielu państwach ustawy antydyskryminacyjne zabraniają traktować ludzi różnie w zależności od pewnych chronionych cech takich jak: pochodzenie etniczne, wiek, płeć, skłonności seksualne, niepełnosprawność lub przekonania religijne. Dozwolone jest analizowanie innych aspektów danych, co się jednak dzieje, jeśli są one skorelowane z chronionymi cechami? Na przykład w regionie, w którym występuje segregacja rasowa, kod pocztowy, a nawet adres IP jest silnym predyktorem przynależności rasowej. W tym kontekście absurdalne jest założenie, że algorytm potrafi na podstawie danych wejściowych obciążonych błędem systematycznym wygenerować uczciwe i bezstronne dane wyjściowe [85]. Jednak to założenie często bywa przyjmowane przez zwolenników podejmowania decyzji na podstawie danych. To nastawienie zostało żartobliwie opisane tak: „uczenie maszynowe jest jak pranie pieniędzy, tylko dotyczy upředzeń” [86].

Systemy analizy prognostycznej jedynie przewidują przyszłość na podstawie przeszłości. Jeśli przeszłość jest naznaczona dyskryminacją, systemy skodyfikują tę dyskryminację. Jeżeli chcemy, by przyszłość była lepsza od przeszłości, konieczna jest wyobraźnia moralna, a wyłącznie ludzie mogą być jej źródłem [87]. Dane i modele powinny być naszymi narzędziami, a nie władcami.

## **Odpowiedzialność i rozliczalność**

Zautomatyzowane podejmowanie decyzji otwiera pytania o odpowiedzialność i rozliczalność [87]. Gdy człowiek popełni błąd, można go za to rozliczyć, a osoba, której dana decyzja dotyczyła, może się

od niej odwołać. Algorytmy też mogą się mylić, kto jednak będzie rozliczany za ich błędy [88]? Kto jest winny, gdy autonomiczny samochód powoduje wypadek? Czy istnieje jakieś rozwiązanie, jeśli automatyczny algorytm oceny zdolności kredytowej regularnie odrzuca wnioski osób określonej rasy lub wyznawców danej religii? Jeżeli decyzja systemu uczenia maszynowego zostanie poddana ocenie sądu, to czy będziesz potrafił wytłumaczyć przed sądem, w jaki sposób algorytm podjął decyzję?

Agencje ratingowe to często podawany przykład zbierania danych do podejmowania decyzji na temat ludzi. Niska ocena kredytowa utrudnia życie, ale zwykle jest oparta na adekwatnych faktach dotyczących historii kredytowej, a błędy w rejestrach można poprawić (choć agencje zazwyczaj tego nie ułatwiają). Jednak algorytmy oceny oparte na uczeniu maszynowym przeważnie wykorzystują dużo bardziej zróżnicowane dane wejściowe i są wysoce niejasne, przez co utrudniają zrozumienie konkretnych decyzji i stwierdzenie, czy ktoś nie został potraktowany w nieuczciwy lub dyskryminujący sposób [89].

Ocena kredytowa podsumowuje to, „jak zachowywałeś się w przeszłości”. Z kolei analityka prognozytyczna zwykle uwzględnia to, „kto jest podobny do Ciebie i jak te osoby zachowywały się w przeszłości”. Porównywanie do innych oznacza stereotypowe traktowanie ludzi — np. na podstawie miejsca zamieszkania (jest to dobre przybliżenie rasy i klasy socjoekonomicznej). Co się dzieje z ludźmi, którzy zostali umieszczeni w nieodpowiedniej kategorii? Ponadto jeśli decyzja jest nieprawidłowa z powodu błędnych danych, odwołanie się od niej jest prawie niemożliwe [87].

Wiele danych ma charakter statystyczny. To oznacza, że nawet jeśli rozkład prawdopodobieństwa dla całej populacji jest prawidłowy, w konkretnych przypadkach decyzje mogą być błędne. Na przykład jeśli średni oczekiwany czas życia w danym kraju to 80 lat, nie oznacza to, że ktoś oczekuje, iż padniesz martwy w dniu 80 urodzin. Na podstawie średniej i rozkładu prawdopodobieństwa nie da się dużo powiedzieć na temat długości życia danej osoby. Podobnie dane wyjściowe w systemie prognozytycznym są probabilistyczne i w konkretnych sytuacjach mogą się okazać błędne.

Ślepa wiara w wyższość podejmowania decyzji na podstawie danych jest nie tylko fałszywa, ale wprost niebezpieczna. Wraz z rozpowszechnianiem się podejmowania decyzji na podstawie danych trzeba ustalić, jak zapewnić rozliczalność i przejrzystość algorytmów, jak uniknąć wzmacniania istniejących uprzedzeń i jak poprawiać systemy, kiedy — co nieuniknione — popełnią błędy.

Trzeba też ustalić, jak zapobiegać używaniu danych do szkodenia ludziom oraz jak wykorzystać pozytywny potencjał tych informacji. Analityka pozwala np. wykrywać finansowe i społeczne aspekty życia ludzi. Z jednej strony można to zastosować do kierowania pomocy i wsparcia do tych osób, które najbardziej tego potrzebują. Z drugiej takie dane są czasem używane przez agresywne firmy do namierzania osób w potrzebie i sprzedawania im ryzykownych produktów takich jak wysokooprocentowane pożyczki i bezwartościowe dyplomy uczelni [87, 90].

## **Pętla sprzężenia zwrotnego**

Nawet w aplikacjach prognozytycznych mających mniej istotny bezpośredni wpływ na ludzi (np. w systemach rekomendacji) pojawiają się trudne kwestie, które trzeba uwzględnić. Gdy usługi zaczynają dobrze prognozować treści, jakie użytkownicy zechcą zobaczyć, mogą wyświetlać same materiały potwierdzające opinie tych osób. Prowadzi to do powstawania „komory rezonansowej”, co może skutkować stereotypami, dezinformacją i polaryzacją poglądów. Wpływ „komór rezonansowych” na kampanie wyborcze już dziś widać w mediach społecznościowych [91].

Gdy analityka prognostyczna wpływa na ludzkie życie, wyjątkowo zgubne problemy pojawiają się z powodu pętli sprzężenia zwrotnego. Pomyśl np. o pracodawcach oceniających potencjalnych pracowników na podstawie oceny zdolności kredytowej. Możliwe, że jesteś dobrym pracownikiem z wysoką oceną zdolności kredytową. Jednak nagle z powodu nieszczęścia, nad którym nie masz kontroli, wpadasz w problemy finansowe. Ponieważ nie płacisz rachunków, Twoja ocena zdolności kredytowej spada i trudniej Ci znaleźć pracę. Brak zatrudnienia wypycha Cię w biedę, co skutkuje jeszcze większym spadkiem oceny zdolności kredytowej, przez co masz jeszcze większe problemy ze zdobyciem posady [87]. Znajdujesz się na równi pochyłej z powodu szkodliwych założeń ukrytych pod płaszczykiem danych i matematycznej ścisłości.

Nie zawsze da się przewidzieć powstawanie takich pętli sprzężenia zwrotnego. Jednak wielu ich konsekwencji można się domyślić, analizując cały system (nie tylko jego skomputeryzowane elementy, ale też powiązanych z nim ludzi). To podejście to *myślenie systemowe* [92]. Możemy próbować zrozumieć, jak system analizy danych reaguje na różne zachowania, struktury lub cechy. Czy system wzmacnia i nasila istniejące różnice między ludźmi (sprawiając, że bogaci stają się jeszcze bogatsi, a biedni — jeszcze biedniejsi), czy prowadzi do zwalczania niesprawiedliwości? Ponadto nawet mając najlepsze intencje, trzeba mieć świadomość niezamierzonych konsekwencji.

## Prywatność i śledzenie

Obok kłopotów z analityką prognostyczną (np. z wykorzystywaniem danych do podejmowania automatycznych decyzji wpływających na ludzi) występują też problemy etyczne związane ze zbieraniem danych. Jak wygląda relacja między organizacjami rejestrującymi dane a osobami, których te dane dotyczą?

Gdy system przechowuje tylko dane bezpośrednio wpisane przez użytkowników, ponieważ ci chcą, aby oprogramowanie przechowywało je i przetwarzało w określony sposób, system świadczy usługę na rzecz użytkownika. Klientem jest wtedy użytkownik. Jeśli jednak aktywność użytkowników jest śledzona i rejestrowana jako efekt uboczny ich innych działań, relacja jest mniej oczywista. Usługa nie robi wtedy tylko tego, co użytkownik jej każe, ale też realizuje interesy jej właścicieli, które mogą być sprzeczne z interesami użytkowników.

Śledzenie danych behawioralnych zyskuje na znaczeniu w komunikujących się z użytkownikami mechanizmach wielu usług internetowych. Śledzenie, które wyniki wyszukiwania zostały kliknięte, pomaga poprawić ich kolejność. Rekomendacje typu „osobom, którym spodobało się X, podobało się też Y” pomagają użytkownikom znajdować ciekawe i użyteczne rzeczy. Testy A/B i analizy wpływu użytkowników mogą pomóc usprawnić interfejs użytkownika. Te cechy wymagają śledzenia zachowania użytkowników, ale przynosi to korzyści tym ostatnim.

Jednak śledzenie często nie kończy się na opisanym poziomie (zależy to od modelu biznesowego firmy). Jeśli usługa zarabia na reklamach, klientami są reklamodawcy, a interesy użytkowników znajdują się na drugim miejscu. Śledzenie danych jest wtedy bardziej szczegółowe, analizy są dalej posunięte, a dane są przechowywane dłużej, aby móc wygenerować szczegółowy profil każdej osoby na potrzeby marketingu.

Wtedy relacja między firmą a użytkownikami, których dane są zbierane, zaczyna wyglądać inaczej. Użytkownik otrzymuje darmową usługę i jest nakłaniany do tego, by jak najczęściej z niej korzystać.



Śledzenie użytkownika służy nie tej osobie, ale reklamodawcom, którzy finansują usługę. Uważam, że takie związki można poprawnie opisać słowem o bardziej złowieszczym wydźwięku — *inwigilacja*.

## Inwigilacja

W ramach eksperymentu myślowego spróbuj zastąpić słowo *dane* wyrazem *inwigilacja* i sprawdź, czy często powtarzane zwroty nadal brzmią równie dobrze [93]. Spójrz na to: „W naszej opartej na inwigilacji organizacji rejestrujemy w czasie rzeczywistym strumienie inwigilacji i zapisujemy je w naszej hurtowni inwigilacji. Nasi specjaliści od inwigilacji posługują się zaawansowanymi analizami i przetwarzaniem na potrzeby inwigilacji w celu wyciągania nowych wniosków”.

Ten eksperyment myślowy jest zaskakująco polemiczny w kontekście tej książki (*Projektowanie aplikacji intensywnie prowadzących inwigilację*). Uważam jednak, że potrzebne są mocne słowa, aby podkreślić znaczenie omawianych tu działań. Próbując sprawić, by oprogramowanie „opanoowało świat” [94], opracowaliśmy najbardziej rozbudowaną infrastrukturę do inwigilacji w historii świata. Pędząc w kierunku internetu rzeczy, błyskawicznie zbliżamy się do powstania świata, w którym każdy zamieszkały obszar zawiera przynajmniej jeden podłączony do internetu mikrofon — w smartfonach, inteligentnych telewizorach, sterowanych głosem urządzeniach podręcznych, elektronicznych nianiach, a nawet zabawkach dla dzieci wykorzystujących oparte na chmurze rozpoznawanie mowy. Doniesienia o poziomie zabezpieczeń w wielu tych urządzeniach są zatrważające [95].

Nawet najbardziej totalitarne i represyjne reżimy mogły tylko marzyć o umieszczeniu mikrofonu w każdym pomieszczeniu i zmuszeniu wszystkich ludzi do ciągłego noszenia urządzeń, które pozwalają śledzić lokalizację i ruchy ich właścicieli. Jednak najwyraźniej dobrowolnie, a nawet z entuzjazmem rzucamy się w świat pełnej inwigilacji. Różnica polega na tym, że dane są zbierane przez korporacje, a nie przez agencje rządowe [96].

Nie wszystkie przypadki zbierania danych można zaklasyfikować jako inwigilację. Jednak analizowanie ich z tej perspektywy pomaga zrozumieć relację użytkowników z jednostką rejestrującą informacje. Dlaczego najwyraźniej z radością akceptujemy inwigilację ze strony korporacji? Możesz uważasz, że nie masz nic do ukrycia. Oznacza to, że w pełni popierasz obecną strukturę władzy, nie należysz do marginalizowanej mniejszości i nie obawiasz się prześladowań [97]. Nie każdy ma tyle szczęścia. Możliwe też, że cel zbierania danych wydaje się niegroźny. Nie służą one wywieraniu otwartego nacisku lub zmuszaniu do uległości, a tylko do generowania trafniejszych rekomendacji i bardziej spersonalizowanych informacji marketingowych. Jednak w połączeniu z oceną analityki prognostycznej z poprzedniego punktu różnice między omawianymi obszarami się zacierają.

Już dostępne są zniżki u ubezpieczycieli samochodów powiązane z umieszczeniem rejestratora w samochodzie, a zakres ubezpieczenia zdrowotnego może zależeć od tego, czy klient nosi urządzenie rejestrujące jego stan. Gdy inwigilacja służy do określania rzeczy mających wpływ na ważne aspekty życia (takie jak zakres ubezpieczenia lub zatrudnienie), przestaje wydawać się niegroźna. Ponadto analizy danych mogą ujawniać zaskakująco poufne informacje. Na przykład czujnik ruchu z inteligentnego zegarka lub rejestratora stanu zdrowia można wykorzystać do całkiem precyzyjnego ustalenia, co pisesz (np. hasła) [98], a algorytmy analityczne są przecież coraz lepsze.

## Zgoda i wolność wyboru

Można stwierdzić, że użytkownicy dobrowolnie posługują się usługami, które śledzą ich aktywność, i zaakceptowali warunki korzystania z usługi i politykę prywatności, tak więc zgadzają się na zbieranie danych. Można nawet twierdzić, że użytkownicy otrzymują cenną usługę w zamian za udostępniane dane. Dlatego śledzenie jest niezbędne, aby udostępniać daną usługę. Sieci społecznościowe, wyszukiwarki i różne inne bezpłatne usługi internetowe są bez wątpienia wartościowe dla użytkowników. Jednak z tym argumentem wiążą się pewne problemy.

Użytkownicy nie wiedzą, jakie dane udostępniają do zapisu w bazie ani w jaki sposób te informacje są zachowywane i przetwarzane. Polityka prywatności częściej komplikuje sytuację, niż ją rozjaśnia. Bez wiedzy o tym, co się dzieje z ich danymi, użytkownicy nie mogą wyrazić prawdziwej zgody. Ponadto często dane od jednego użytkownika zawierają informacje o innych osobach, które nie są użytkownikami danej usługi i nie akceptowały żadnych warunków. Omawiane w tej części książki pochodne zbiory danych (w których informacje z całej bazy użytkowników można łączyć z danymi z procesu śledzenia i zewnętrznymi źródłami) są właśnie czymś, czego użytkownicy nie mogą w pełni zrozumieć.

Ponadto dane są pobierane od użytkowników w jednokierunkowym procesie. Nie występuje tu prawdziwie wzajemna relacja ani uczciwa wymiana wartości. Nie ma tu dialogu, a użytkownicy nie mogą negocjować, ile danych chcą udostępniać i jakie usługi otrzymają w zamian. Relacja między właścicielem usługi a użytkownikiem jest wysoce asymetryczna i jednostronna. Warunki ustala właściciel usługi, a nie użytkownik [99].

Użytkownik, który nie chce się zgodzić na inwigilację, ma tak naprawdę tylko jedną możliwość — nie korzystać z usługi. Jednak ten wybór nie jest wolny od kosztów. Jeśli usługa stała się tak popularna, że „jest uznawana przez większość osób za niezbędną do podstawowego życia społecznego” [99], nierozsądnie byłoby oczekiwać, że ludzie z niej zrezygnują. Korzystanie z niej jest w praktyce konieczne. Na przykład w większości społeczności zachodniego świata normą stało się noszenie smartfona, używanie Facebooka do utrzymywania kontaktów towarzyskich i posługiwanie się wyszukiwarką Google do znajdowania informacji. Zwłaszcza gdy usługa służy do budowania sieci międzyludzkich, *rezygnacja* z niej wiąże się z kosztami społecznymi.

Odmowa korzystania z usługi dlatego, że śledzi ona użytkowników, jest możliwa tylko w przypadku niewielkiej grupy osób, które mają wystarczająco dużo czasu i wiedzy, aby zrozumieć politykę prywatności, oraz mogą sobie pozwolić na rezygnację z życia społecznego lub okazji zawodowych dostępnych dzięki danej usłudze. Osoby w mniej komfortowej sytuacji nie mają takiej wolności wyboru. Inwigilacja jest w ich przypadku nieunikniona.

## Prywatność i wykorzystanie danych

Czasem ludzie twierdzą, że „prywatność jest martwa”. Argumentują to tym, że niektórzy użytkownicy zamieszczają w mediach społecznościowych wszelkie możliwe informacje na temat swojego życia — nieraz nudne i czasem bardzo osobiste. Jednak to twierdzenie jest nieprawdziwe i wynika z błędnego rozumienia słowa *prywatność*.

Zachowywanie prywatności nie oznacza, że wszystko trzeba trzymać w sekrecie. Prywatność wynika ze swobody wyboru tego, co komu ujawnić, co upublicznić, a co zachować dla siebie. Prawo

do prywatności jest prawem podejmowania decyzji. Umożliwia każdemu stwierdzenie w każdej sytuacji, jaką pozycję przyjąć na skali jawność – niejawność [99]. Jest to ważny aspekt wolności i autonomii człowieka.

Gdy dane na temat ludzi są zbierane za pomocą infrastruktury służącej do inwigilacji, oznacza to nie tyle łamanie prawa do prywatności, ile przekazanie go jednostce rejestrującej te informacje. Firmy zbierające dane w pewnym sensie mówią: „Zaufajcie, że wykorzystamy wasze dane we właściwy sposób”. To oznacza, że prawo decydowania o tym, co ujawnić, a co zachować w sekrecie, jest cedowane z poszczególnych osób na organizację.

Firmy mogą zdecydować się nie upowszechniać większości poufnych danych uzyskanych dzięki inwigilacji, ponieważ ujawnianie ich byłoby postrzegane jako okropne i mogłoby zaszkodzić modelowi biznesowemu (który polega na tym, aby wiedzieć o ludziach więcej niż inne organizacje). Szczegółowe informacje na temat użytkowników są ujawniane tylko pośrednio — np. w formie narzędzi do kierowania reklam do specyficznych grup osób (przykładowo cierpiących na konkretną chorobę).

Nawet jeśli poszczególnych użytkowników nie da się osobiście zidentyfikować w grupie, do której kierowana jest dana reklama, osoby te tracą wybór w kwestii ujawniania poufnych informacji takich jak konkretna przypadłość, na którą cierpią. To nie użytkownik decyduje na podstawie własnych preferencji o tym, co jest komuś ujawniane. To firma wykorzystuje kontrolę nad prawem do prywatności, aby zmaksymalizować zysk.

Wiele firm nie chce być *postrzeganych* jako okropne. Te organizacje unikają pytań na temat tego, jak inwazyjny jest stosowany przez nie proces zbierania danych, zamiast tego koncentrują się na zarządzaniu wizerunkiem. A nawet samo zarządzanie wizerunkiem często słabo im wychodzi. Na przykład coś może być prawdą, ale powoduje przykre wspomnienia i użytkownik nie chce, aby mu o tym przypominać [100]. W przypadku każdego danych należy oczekiwać, że mogą zostać wykorzystane w błędny, niepożądany lub niewłaściwy sposób. Dlatego trzeba zbudować mechanizmy do radzenia sobie z takimi problemami. To, czy coś jest „niepożądane” lub „niewłaściwe”, sprowadza się oczywiście do ludzkiego osądu. Algorytmy nie są świadome takich kwestii, chyba że bezpośrednio zaprogramujemy w nich szacunek do ludzkich potrzeb. Jako inżynierowie systemów musimy zachować pokorę, akceptując problemy i planując radzenie sobie z nimi.

Ustawienia prywatności pozwalające użytkownikowi usługi internetowej kontrolować, które aspekty ich danych są widoczne dla innych osób, to punkt wyjścia do zwrócenia części władzy ludziom. Jednak niezależnie od ustawień właściciel usługi nadal ma pełny dostęp do danych i może je wykorzystać w dowolny sposób dozwolony według polityki prywatności. Nawet jeśli firma obiecuje, że nie będzie sprzedawać danych trzeciej stronie, zwykle zastrzega sobie nieograniczone prawo do wewnętrznego przetwarzania i analizowania danych, często posuwając się w tym obszarze znacznie dalej, niż jest to widoczne dla użytkowników.

Takie zakrojone na dużą skalę przeniesienie prawa do kontrolowania prywatności z jednostek na korporacje jest czymś do tej pory niespotykanym [99]. Inwigilacja istnieje od zawsze, jednak w przeszłości była kosztowna i ręczna, a nie skalowalna i zautomatyzowana. Relacje oparte na zaufaniu też występowały — np. między pacjentem a lekarzem lub oskarżonym a jego adwokatem — jednak w takich sytuacjach wykorzystywanie danych było ściśle regulowane ograniczeniami etycznymi,

prawnymi i ustawowymi. Usługi internetowe znacznie ułatwiły rejestrowanie olbrzymich ilości poufnych informacji bez opartej na zrozumieniu zgody i wykorzystywanie ich na masową skalę bez wiedzy użytkowników o tym, co się dzieje z ich prywatnymi danymi.

## Dane jako zasoby i źródło władzy

Ponieważ dane behawioralne są produktem ubocznym interakcji użytkowników z usługą, czasem nazywa się je „spalinami” (ang. *data exhaust*), co sugeruje, że dane są bezwartościowym odpadem. Z tej perspektywy analitykę behawioralną i prognostyczną można postrzegać jako formę recyklingu pozwalającą wydobyć wartość z danych, które w przeciwnym razie zostałyby usunięte.

Jednak poprawniejszy jest odwrotny punkt widzenia. Z perspektywy ekonomicznej jest tak, że jeśli to ukierunkowana reklama zapewnia dochody właścicielom usługi, dane behawioralne na temat użytkowników są podstawowym zasobem firmy. Aplikacja, z której użytkownik korzysta, stanowi tylko sposób na nakłonienie użytkowników do udostępniania coraz większej ilości danych służącej do inwigilacji infrastruktury [99]. Wspaniała kreatywność człowieka i relacje społeczne, które często stanowią podstawę usług internetowych, są cynicznie wykorzystywane przez maszynę do wydobywania danych.

Twierdzenie, że dane osobowe są cennym zasobem, jest poparte istnieniem handlarzy danymi. Są to działające niejawnie podejrzane firmy, które skupują, agregują, analizują, przetwarzają i odsprzedają poufne dane osobowe na temat ludzi, najczęściej w celach marketingowych [90]. Wartość startupów jest oceniana na podstawie liczby użytkowników („oczu”), czyli możliwości z zakresu inwigilacji.

Ponieważ dane są cenne, wiele osób chce je zdobyć. Oczywiście zależy na nim firmom. To właśnie dlatego organizacje zbierają dane. Jednak także rządy próbują je uzyskać — za pomocą tajnych umów, przymusu prawnego lub zwykłej kradzieży [101]. Gdy firma bankrutuje, zebrane przez nią dane osobowe są jednym z zasobów, które zostają sprzedane. Ponadto dane trudno jest zabezpieczyć, dlatego ich ujawnienie zdarza się niepokojąco często [102].

Te spostrzeżenia sprawiły, że krytycy twierdzą, iż dane są nie tylko zasobami, ale nawet „toksycznymi zasobami” [101], a przynajmniej „niebezpiecznymi materiałami” [103]. Nawet jeśli sądzisz, że potrafisz zapobiec nadużyciom w związku z danymi, musisz zrównoważyć korzyści z dostępu do nich z ryzykiem dostania się informacji w niepowołane ręce. Systemy komputerowe mogą zostać złamane przez przestępców lub wrogie służby wywiadu, dane mogą zostać ujawnione przez pracowników organizacji, władzę w firmie mogą przejąć pozbawieni skrupułów menedżerowie niepodzielający Twoich wartości, a rządy w państwie mogą trafić w ręce reżimu, który nie będzie miał skrupułów przed zmuszaniem firm do przekazywania mu danych.

W trakcie zbierania danych trzeba uwzględnić nie tylko aktualne środowisko polityczne, ale też wszystkie możliwe przyszłe rządy. Nie ma gwarancji, że każdy wybrany w przyszłości rząd będzie przestrzegał praw człowieka i swobód obywatelskich, dlatego „instalowanie technologii, które mogą kiedyś wspierać państwo policyjne, jest postępowaniem mało obywatelskim” [104].

„Wiedza to władza” — mówi stare przysłowie. Ponadto „kontrolowanie innych przy jednoczesnym unikaniu bycia kontrolowanym to jedna z najważniejszych form władzy” [105]. To dlatego rządy totalitarne chcą inwigilować innych. Umożliwia im to kontrolowanie społeczeństwa. Choć dzisiejsze

firmy informatyczne nie starają się skrycie zdobyć władzy politycznej, gromadzone przez nie dane i wiedza dają im znaczną kontrolę nad naszym życiem. Duża jej część jest ukryta i znajduje się poza publicznym nadzorem [106].

### **Pamiętając o rewolucji przemysłowej...**

Dane są cechą definiującą erę informacji. Internet, składowanie danych, przetwarzanie i automatyzacja oparta na oprogramowaniu mają istotny wpływ na globalną ekonomię i społeczeństwo. Nasze życie codzienne i organizacja społeczna zmieniły się w ostatnim dziesięcioleciu oraz zapewne będą podlegać radykalnym przemianom także w najbliższych dekadach. Przywodzi to na myśl porównania do rewolucji przemysłowej [87, 96].

Rewolucja przemysłowa nastąpiła w wyniku istotnych osiągnięć w obszarze technologii i rolnictwa. Spowodowała trwały wzrost ekonomiczny i w długiej perspektywie znacznie poprawiła standard życia ludzi. Jednak były z nią związane poważne problemy — katastrofalne zanieczyszczenie powietrza (z powodu dymu i procesów chemicznych) oraz wody (odpadami przemysłowymi i ludzkimi). Właściciele fabryk opływali w bogactwa, podczas gdy robotnicy mieszkali bardzo ubogo i pracowali przez wiele godzin w złych warunkach. Powszechne było zatrudnianie dzieci — także na niebezpiecznych i słabo płatnych stanowiskach w kopalniach.

Minęło dużo czasu do wprowadzenia zabezpieczeń takich jak regulacje chroniące środowisko, protokoły bezpieczeństwa dla miejsc pracy, zakaz pracy nieletnich i kontrole żywności. Bez wątpienia koszty prowadzenia działalności wzrosły, ponieważ firmy nie mogą już wylewać odpadów do rzek, sprzedawać zepsutego jedzenia i wykorzystywać pracowników. Jednak społeczeństwo jako całość bardzo na tym zyskało i niewiele osób chciałoby wrócić do okresu sprzed wprowadzenia regulacji [87].

Podobnie jak rewolucja przemysłowa miała swoją ciemną stronę, z którą trzeba było sobie poradzić, tak i przejście do ery informacji wiąże się z poważnymi problemami. Musimy się z nimi skonfrontować i je rozwiązać. Wierzę, że jednym z tych problemów jest rejestrowanie i wykorzystywanie danych. Oto słowa Bruce’a Schneiera [96]:

W erze informacji dane to odpowiednik zanieczyszczenia, a ochrona prywatności jest jak wyzwanie związane z ochroną środowiska. Prawie wszystkie komputery generują informacje. Te dane zalegają i gniją. To, jak sobie z tym poradzimy (jak będziemy zapobiegać rozprzestrzenianiu się ich i jak będziemy się ich pozbywać), ma istotny wpływ na stan naszej informacyjnej ekonomii. Podobnie jak dziś spoglądamy na pierwsze dekady ery przemysłowej i zastanawiamy się, jak nasi przodkowie mogli zignorować zanieczyszczenie w gorączkowym budowaniu świata przemysłowego, tak nasze wnuki będą patrzeć na nas w pierwszych dekadach ery informacji i oceniać nas pod kątem tego, jak poradziliśmy sobie z wyzwaniami z obszaru rejestrowania i niewłaściwego wykorzystywania danych.

Powinniśmy starać się o to, by nasze wnuki były z nas dumne.

### **Ustawodawstwo i samoregulacja**

Prawa nastawione na ochronę danych mogą pomóc w ochronie praw jednostki. Na przykład w dyrektywie dotyczącej ochrony danych z 1995 r. (*European Data Protection Directive*) napisano, że dane osobowe muszą być „zbierane w określonych, bezpośrednio podanych i zgodnych z prawem celach oraz nie mogą być dodatkowo przetwarzane w sposób niezgodny z tymi celami”, a ponadto muszą być „adekwatne, właściwe i dostosowane zakresem względem celów, dla których są rejestrowane” [107].

Jednak można wątpić, czy ta dyrektywa jest skuteczna w kontekście dzisiejszego internetu [108]. Zawarte w niej zasady są bezpośrednio sprzeczne z filozofią Big Data, która polega na maksymalizowaniu zbierania danych, łączeniu ich z innymi zbiorami, eksperymentowaniu i eksplorowaniu informacji, aby uzyskać nowe wnioski. Eksploracja oznacza wykorzystywanie danych do nieprzewidywanych wcześniej zadań, co jest odwrotnością „określonych i bezpośrednio podanych” celów, jakie użytkownik zaakceptował (jeśli w ogóle można mówić o świadomym wyrażaniu zgody [109]). Obecnie trwają prace nad nowymi regulacjami [89].

Firmy rejestrujące duże ilości danych sprzeciwiają się regulacjom, twierdząc, że stanowią one obciążenie i hamują innowacyjność. W pewnym zakresie ten sprzeciw jest usprawiedliwiony. Na przykład udostępnianie danych medycznych skutkuje oczywistym ryzykiem naruszenia prywatności, jednak jest też szansa. Ilu zgonom można będzie zapobiec, jeśli analiza danych pomoże w usprawnieniu diagnostyki lub znalezieniu lepszych sposobów leczenia [110]? Nadmierna regulacja może utrudniać dokonanie przełomu. Trudno jest zrównoważyć potencjalne szanse z zagrożeniami [105].

Zasadniczo uważam, że w branży informatycznej potrzebujemy zmiany kulturowej dotyczącej danych osobowych. Powinniśmy przestać traktować użytkowników jak wskaźników, które należy zoptymalizować. Musimy pamiętać, że użytkownicy to ludzie zasługujący na szacunek, godność i zachowanie kontroli. Powinniśmy wprowadzić samoregulacje w obszarze zbierania i przetwarzania danych, aby uzyskać i utrzymać zaufanie ludzi, którzy polegają na naszym oprogramowaniu [111]. Musimy też podjąć trud edukowania użytkowników końcowych na temat tego, jak ich dane są wykorzystywane, zamiast utrzymywać ich w nieświadomości.

Powinniśmy umożliwić każdej osobie zachowanie prywatności (czyli kontroli nad własnymi danymi) i nie odbierać możliwości podejmowania decyzji za pomocą inwigilacji. Jednostkowe prawo do kontrolowania własnych danych jest jak środowisko naturalne w parku narodowym. Jeśli nie będziemy aktywnie go chronić i dbać o nie, zostanie zniszczone. Będzie to tragedią dla społeczeństwa i zaszkodzi wszystkim. Powszechna inwigilacja nie jest nieunikniona. Jeszcze możemy ją powstrzymać.

To, w jaki sposób jesteśmy w stanie to osiągnąć, stanowi otwartą kwestię. Zaczniemy od tego, że nie powinniśmy przechowywać danych w nieskończoność, tylko musimy je usuwać, gdy nie są już potrzebne [111, 112]. Usuwanie danych jest sprzeczne z zasadą niemodyfikowalności (zob. punkt „Ograniczenia niemodyfikowalności”), jednak ten problem można rozwiązać. Obiecującym podejściem jest wymuszanie kontroli dostępu z użyciem protokołów kryptograficznych, a nie za pomocą samej polityki [113, 114]. Na ogólnym poziomie niezbędne będą zmiany kultury i nastawienia.

## Podsumowanie

W tym rozdziale opisano nowe sposoby projektowania systemów danych. Przedstawiłem tu własne opinie i spekulacje na temat przyszłości. Zacząłem od spostrzeżenia, że nie istnieje jedno narzędzie, które wydajnie sobie radzi we wszystkich scenariuszach. Dlatego aplikacje muszą łączyć kilka narzędzi, aby wykonywać stawiane im zadania. Opisałem, jak za pomocą przetwarzania wsadowego i strumieni zdarzeń rozwiązać problem *integrowania danych* i umożliwić przepływ danych między różnymi systemami.

W opisanym podejściu różne narzędzia są traktowane jako system zapisu, na podstawie którego za pomocą transformacji uzyskiwane są inne dane. W ten sposób można zarządzać indeksami, widokami zmaterializowanymi, modelami w uczeniu maszynowym, podsumowaniami statystycznymi itd. Dzięki temu, że generowanie i transformowanie danych odbywa się asynchronicznie i za pomocą luźno powiązanych komponentów, problem z jednego obszaru nie rozprzestrzenia się na niepowiązane części systemu, co zwiększa niezawodność i odporność na błędy całego systemu.

Przedstawianie przepływu danych jako transformacji jednego zbioru danych w inny pomaga też w modyfikowaniu aplikacji. Jeśli chcesz zmienić jeden z etapów przetwarzania (np. w celu zmodyfikowania struktury indeksu lub pamięci podręcznej), możesz ponownie uruchomić nowy kod odpowiedzialny za transformację dla całego wejściowego zbioru danych. Pozwala to jeszcze raz wygenerować dane wyjściowe. Podobnie jeśli wystąpi błąd, możesz poprawić kod i ponownie przetworzyć dane, aby przywrócić stan.

Opisane procesy są podobne do tego, jak bazy obecnie działają wewnętrznie. Można więc przedstawić pomysł aplikacji opartych na przepływie danych jako *podział* bazy na komponenty i budowanie aplikacji poprzez łączenie luźno powiązanych elementów.

Stan pochodny można aktualizować na podstawie obserwowania zmian w danych podstawowych. Ponadto sam stan pochodny może być obserwowany przez konsumentów z dalszych etapów przetwarzania. Ten przepływ danych można rozciągnąć aż do urządzenia użytkownika końcowego, które wyświetla dane, i budować w ten sposób dynamicznie aktualizowane interfejsy użytkownika odzwierciedlające zmiany w danych i działające także w trybie offline.

Dalej opisano, jak zapewnić poprawność przetwarzania w obliczu błędów. Zobaczyłeś, że można w skalowalny sposób zaimplementować mocne gwarancje integralności za pomocą asynchronicznego przetwarzania zdarzeń. Wymaga to zastosowania identyfikatorów operacji na trasie między punktami końcowymi (w celu zapewnienia idempotencji działań) i asynchronicznego sprawdzania więzów. Klienci mogą albo oczekiwać do czasu sprawdzenia więzów, albo kontynuować pracę, ryzykując, że trzeba będzie przeprosić za naruszenie więzów. Przedstawione rozwiązanie jest bardziej skalowalne i niezawodne niż tradycyjne podejście (czyli używanie transakcji rozproszonych). Jest też zgodne z tym, w jaki sposób wiele procesów biznesowych funkcjonuje w praktyce.

Dzięki budowaniu struktury aplikacji na podstawie przepływu danych i dzięki asynchronicznemu sprawdzaniu więzów można uniknąć większości zadań związanych z koordynacją i tworzyć systemy, które zachowują integralność, a jednocześnie działają wydajnie — nawet gdy są rozproszone geograficznie i w obliczu błędów. Dalej opisano przeprowadzanie audytów w celu sprawdzania integralności danych i wykrywania uszkodzeń.

Ostatni fragment to spojrzenie z perspektywy i omówienie wybranych etycznych aspektów budowania aplikacji intensywnie przetwarzających dane. Zobaczyłeś, że choć dane można stosować w dobrym celu, czasem powodują one szkody: gdy są używane do podejmowania decyzji poważnie wpływających na życie ludzi i trudnych do podważenia, gdy prowadzą do dyskryminacji, wyzysku, racjonalizowania inwigilacji i udostępniania poufnych informacji. Istnieje też ryzyko wycieków danych, a ponadto może się okazać, że korzystanie z danych w dobrych intencjach będzie miało niezamierzone konsekwencje.

Ponieważ oprogramowanie i dane mają tak duży wpływ na świat, inżynierowie muszą pamiętać, że są odpowiedzialni za budowanie takiej rzeczywistości, w jakiej chcą żyć — takiej, w której ludzie są traktowani humanitarnie i z szacunkiem. Mam nadzieję, że będziemy wspólnie dążyć do tego celu.

## **Literatura cytowana**

- [1] Rachid Belaid, *Postgres Full-Text Search is Good Enough!*, *rachbelaid.com*, 13 lipca 2015 (<http://rachbelaid.com/postgres-full-text-search-is-good-enough/>).
- [2] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar i in., *Challenges to Adopting Stronger Consistency at Scale*, w: „15th USENIX Workshop on Hot Topics in Operating Systems” (HotOS), maj 2015 (<https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf>).
- [3] Pat Helland i Dave Campbell, *Building on Quicksand*, w: „4th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2009 ([https://database.cs.wisc.edu/cidr/cidr2009/Paper\\_133.pdf](https://database.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf)).
- [4] Jessica Kerr, *Provenance and Causality in Distributed Systems*, *blog.jessitron.com*, 25 wrzesień 2016 (<http://blog.jessitron.com/2016/09/provenance-and-causality-in-distributed.html>).
- [5] Kostas Tzoumas, *Batch Is a Special Case of Streaming*, *data-artisans.com*, 15 września 2015 (<https://data-artisans.com/blog/batch-is-a-special-case-of-streaming>).
- [6] Shinji Kim i Robert Blafford, *Stream Windowing Performance Analysis: Concord and Spark Streaming*, *concord.io*, 6 lipca 2016 ([http://concord.io/posts/windowing\\_performance\\_analysis\\_w\\_spark\\_streaming](http://concord.io/posts/windowing_performance_analysis_w_spark_streaming)).
- [7] Jay Kreps, *The Log: What Every Software Engineer Should Know About RealTime Data’s Unifying Abstraction*, *engineering.linkedin.com*, 16 grudnia 2013 (<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>).
- [8] Pat Helland, *Life Beyond Distributed Transactions: An Apostate’s Opinion*, w: „3rd Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2007 (<http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf>).
- [9] *Great Western Railway (1835–1948)*, Network Rail Virtual Archive, *networkrail.co.uk* (<https://www.networkrail.co.uk/running-the-railway/our-routes/western/great-western-mainline/>).
- [10] Jacqueline Xu, *Online Migrations at Scale*, *stripe.com*, 2 lutego 2017 (<https://stripe.com/blog/online-migrations>).
- [11] Molly Bartlett Dishman i Martin Fowler, *Agile Architecture*, w: „O’Reilly Software Architecture Conference”, marzec 2015 (<https://conferences.oreilly.com/software-architecture/sa2015/public/schedule/detail/40388>).
- [12] Nathan Marz i James Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*, Manning, 2015, ISBN: 978-1-617-29034-3 (<https://www.manning.com/books/big-data>).



- [13] Oscar Boykin, Sam Ritchie, Ian O’Connell i Jimmy Lin, *Summingbird: A Framework for Integrating Batch and Online MapReduce Computations*, w: „40th International Conference on Very Large Data Bases” (VLDB), wrzesień 2014 (<http://www.vldb.org/pvldb/vol7/p1441-boykin.pdf>).
- [14] Jay Kreps, *Questioning the Lambda Architecture*, *oreilly.com*, 2 lipca 2014 (<https://www.oreilly.com/ideas/questioning-the-lambda-architecture>).
- [15] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps i in., *Liquid: Unifying Nearline and Offline Big Data Integration*, w: „7th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2015 ([https://lsds.doc.ic.ac.uk/sites/default/files/CIDR15\\_Paper25u\\_0.pdf](https://lsds.doc.ic.ac.uk/sites/default/files/CIDR15_Paper25u_0.pdf)).
- [16] Dennis M. Ritchie i Ken Thompson, *The UNIX Time-Sharing System*, „Communications of the ACM”, rocznik 17, nr 7, s. 365 – 375, lipiec 1974 (<https://people.eecs.berkeley.edu/~brewer/cs262/unix.pdf>; <https://dl.acm.org/citation.cfm?doid=361011.361061>).
- [17] Eric A. Brewer i Joseph M. Hellerstein, *CS262a: Advanced Topics in Computer Systems*, notatki z wykładu, University of California, Berkeley, *cs.berkeley.edu*, sierpień 2011 (<http://people.eecs.berkeley.edu/~brewer/cs262/systemr.html>).
- [18] Michael Stonebraker, *The Case for Polystores*, *wp.sigmod.org*, 13 lipca 2015 (<http://wp.sigmod.org/?p=1629>).
- [19] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker i in., *The BigDAWG Polystore System*, „ACM SIGMOD Record”, rocznik 44, nr 2, s. 11 – 16, czerwiec 2015 (<https://homes.cs.washington.edu/~magda/papers/duggan-sigrec15.pdf>; <https://dl.acm.org/citation.cfm?doid=2814710.2814713>).
- [20] Patrycja Dybka, *Foreign Data Wrappers for PostgreSQL*, *vertabelo.com*, 24 marca 2015 (<http://www.vertabelo.com/blog/technical-articles/foreign-data-wrappers-for-postgresql>).
- [21] David B. Lomet, Alan Fekete, Gerhard Weikum i Mike Zwillig, *Unbundling Transaction Services in the Cloud*, w: „4th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2009 (<https://www.microsoft.com/en-us/research/publication/unbundling-transaction-services-in-the-cloud/>).
- [22] Martin Kleppmann i Jay Kreps, *Kafka, Samza and the Unix Philosophy of Distributed Data*, „IEEE Data Engineering Bulletin”, rocznik 38, nr 4, s. 4 – 14, grudzień 2015 (<http://martin.kleppmann.com/papers/kafka-debull15.pdf>).
- [23] John Hugg, *Winning Now and in the Future: Where VoltDB Shines*, *voltdb.com*, 23 marca 2016 (<https://www.voltdb.com/blog/2016/03/23/winning-now-future-voltdb-shines/>).
- [24] Frank McSherry, Derek G. Murray, Rebecca Isaacs i Michael Isard, *Differential Dataflow*, w: „6th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2013 ([http://cidrdb.org/cidr2013/Papers/CIDR13\\_Paper111.pdf](http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf)).
- [25] Derek G. Murray, Frank McSherry, Rebecca Isaacs i in., *Naiad: A Timely Dataflow System*, w: „24th ACM Symposium on Operating Systems Principles” (SOSP), s. 439 – 455, październik 2013 (<http://sigops.org/sosp/sosp13/papers/p439-murray.pdf>; <https://dl.acm.org/citation.cfm?doid=2517349.2522738>).

- [26] Gwen Shapira, *We have a bunch of customers who are implementing »database inside-out« concept and they all ask „is anyone else doing it? are we crazy?“*, twitter.com, 28 lipca 2016 (<https://twitter.com/gwenshap/status/758800071110430720>).
- [27] Martin Kleppmann, *Turning the Database Inside-out with Apache Samza*, w: „Strange Loop“, wrzesień 2014 (<http://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html>).
- [28] Peter Van Roy i Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004, ISBN: 978-0-262-22069-9 (<https://mitpress.mit.edu/books/concepts-techniques-and-models-computer-programming>).
- [29] Juttle Documentation, [juttle.github.io](http://juttle.github.io/juttle/), 2016 (<http://juttle.github.io/juttle/>).
- [30] Evan Czaplicki i Stephen Chong, *Asynchronous Functional Reactive Programming for GUIs*, w: „34th ACM SIGPLAN Conference on Programming Language Design and Implementation” (PLDI), czerwiec 2013 (<http://people.seas.harvard.edu/~chong/pubs/pldi13-elm.pdf>; <https://dl.acm.org/citation.cfm?doid=2491956.2462161>).
- [31] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx i Wolfgang de Meuter. *A Survey on Reactive Programming*, „ACM Computing Surveys”, rocznik 45, nr 4, s. 1 – 34, sierpień 2013 (<http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf>; <https://dl.acm.org/citation.cfm?doid=2501654.2501666>).
- [32] Peter Alvaro, Neil Conway, Joseph M. Hellerstein i William R. Marczak, *Consistency Analysis in Bloom: A CALM and Collected Approach*, w: „5th Biennial Conference on Innovative Data Systems Research” (CIDR), styczeń 2011 (<https://people.eecs.berkeley.edu/~palvaro/cidr11.pdf>).
- [33] Felienne Hermans, *Spreadsheets Are Code*, w: „Code Mesh”, listopad 2015 (<https://vimeo.com/145492419>).
- [34] Dan Bricklin i Bob Frankston, *VisiCalc: Information from Its Creators*, [danbricklin.com](http://danbricklin.com) (<http://danbricklin.com/visicalc.htm>).
- [35] D. Sculley, Gary Holt, Daniel Golovin i in., *Machine Learning: The HighInterest Credit Card of Technical Debt*, w: „NIPS Workshop on Software Engineering for Machine Learning” (SE4ML), grudzień 2014 (<https://research.google.com/pubs/pub43146.html>).
- [36] Peter Bailis, Alan Fekete, Michael J. Franklin i in., *Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity*, w: „ACM International Conference on Management of Data” (SIGMOD), czerwiec 2015 (<http://www.bailis.org/papers/feral-sigmod2015.pdf>; <https://dl.acm.org/citation.cfm?doid=2723372.2737784>).
- [37] Guy Steele, *Re: Need for Macros (Was Re: Icon)*, e-mail na liście dyskusyjnej *ll1-discuss*, [people.csail.mit.edu](http://people.csail.mit.edu), 24 grudnia 2001 (<https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg01134.html>).
- [38] David Gelernter, *Generative Communication in Linda*, „ACM Transactions on Programming Languages and Systems” (TOPLAS), rocznik 7, nr 1, s. 80 – 112, styczeń 1985 (<http://cseweb.ucsd.edu/groups/csag/html/teaching/cse291s03/Readings/p80-gelernter.pdf>; <https://dl.acm.org/citation.cfm?doid=2363.2433>).

- [39] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui i Anne-Marie Kermarrec, *The Many Faces of Publish/Subscribe*, „ACM Computing Surveys”, rocznik 35, nr 2, s. 114 – 131, czerwiec 2003 (<http://www.cs.ru.nl/~pieter/oss/manyfaces.pdf>; <https://dl.acm.org/citation.cfm?doid=857076.857078>).
- [40] Ben Stopford, *Microservices in a Streaming World*, w: „QCon London”, marzec 2016 (<https://www.infoq.com/presentations/microservices-streaming>).
- [41] Christian Posta, *Why Microservices Should Be Event Driven: Autonomy vs Authority*, [blog.christianposta.com](http://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/), 27 maja 2016 (<http://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/>).
- [42] Alex Feyerke, *Say Hello to Offline First*, [hood.ie](http://hood.ie/blog/say-hello-to-offline-first.html), 5 listopada 2013 (<http://hood.ie/blog/say-hello-to-offline-first.html>).
- [43] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko i Manuel Fähndrich, *Global Sequence Protocol: A Robust Abstraction for Replicated Shared State*, w: „29th European Conference on Object-Oriented Programming” (ECOOP), lipiec 2015 (<http://drops.dagstuhl.de/opus/volltexte/2015/5238/>).
- [44] Mark Soper, *Clearing Up React Data Management Confusion with Flux, Redux, and Relay*, [medium.com](https://medium.com/@marksoper/clearing-up-react-data-management-confusion-with-flux-redux-and-relay-aad504e63cae), 3 grudnia 2015 (<https://medium.com/@marksoper/clearing-up-react-data-management-confusion-with-flux-redux-and-relay-aad504e63cae>).
- [45] Eno Thereska, Damian Guy, Michael Noll i Neha Narkhede, *Unifying Stream Processing and Interactive Queries in Apache Kafka*, [confluent.io](http://confluent.io), 26 października 2016 (<https://www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/>).
- [46] Frank McSherry, *Dataflow as Database*, [github.com](https://github.com/frankmcsherry/blog/blob/master/posts/2016-07-17.md), 17 lipca 2016 (<https://github.com/frankmcsherry/blog/blob/master/posts/2016-07-17.md>).
- [47] Peter Alvaro, *I See What You Mean*, w: „Strange Loop”, wrzesień 2015 (<https://www.youtube.com/watch?v=R2Aa4PivG0g>).
- [48] Nathan Marz, *Trident: A High-Level Abstraction for Realtime Computation*, [blog.twitter.com](http://blog.twitter.com), 2 sierpnia 2012 ([https://blog.twitter.com/engineering/en\\_us/a/2012/trident-a-high-level-abstraction-for-realtime-computation.html](https://blog.twitter.com/engineering/en_us/a/2012/trident-a-high-level-abstraction-for-realtime-computation.html)).
- [49] Edi Bice, *Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends*, w: „Merchant Risk Council MRC Vegas Conference”, marzec 2016 (<https://www.slideshare.net/edibice/extremely-low-latency-web-scale-fraud-prevention-with-apache-samza-kafka-and-friends>).
- [50] Charity Majors, *The Accidental DBA*, [charity.wtf](http://charity.wtf), 2 października 2016 (<https://charity.wtf/2016/10/02/the-accidental-dba/>).
- [51] Arthur J. Bernstein, Philip M. Lewis i Shiyong Lu, *Semantic Conditions for Correctness at Different Isolation Levels*, w: „16th International Conference on Data Engineering” (ICDE), luty 2000 (<http://db.cs.berkeley.edu/cs286/papers/isolation-icde2000.pdf>; <http://ieeexplore.ieee.org/document/839387/>).

- [52] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham i S. Sudarshan, *Automating the Detection of Snapshot Isolation Anomalies*, w: „33rd International Conference on Very Large Data Bases” (VLDB), wrzesień 2007 (<http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf>).
- [53] Kyle Kingsbury, seria artykułów poświęconych platformie Jepsen, *aphyr.com*, 2013 – 2016 (<https://aphyr.com/tags/jepsen>).
- [54] Michael Jouravlev, *Redirect After Post*, *theserverside.com*, 1 sierpnia 2004 (<http://www.theserverside.com/news/1365146/Redirect-After-Post>).
- [55] Jerome H. Saltzer, David P. Reed i David D. Clark, *End-to-End Arguments in System Design*, „ACM Transactions on Computer Systems”, rocznik 2, nr 4, s. 277 – 288, październik 1984 (<http://www.ece.drexel.edu/courses/ECE-C631-501/SalRee1984.pdf>; <https://dl.acm.org/citation.cfm?doid=357401.357402>).
- [56] Peter Bailis, Alan Fekete, Michael J. Franklin i in., *Coordination-Avoiding Database Systems*, „Proceedings of the VLDB Endowment”, rocznik 8, nr 3, s. 185 – 196, listopad 2014 (<https://arxiv.org/pdf/1402.2237.pdf>).
- [57] Alex Yarmula, *Strong Consistency in Manhattan*, *blog.twitter.com*, 17 marca 2016 ([https://blog.twitter.com/engineering/en\\_us/a/2016/strong-consistency-in-manhattan.html](https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.html)).
- [58] Douglas B Terry, Marvin M Theimer, Karin Petersen i in., *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, w: „15th ACM Symposium on Operating Systems Principles” (SOSP), s. 172 – 182, grudzień 1995 (<http://css.csail.mit.edu/6.824/2014/papers/bayou-conflicts.pdf>; <https://dl.acm.org/citation.cfm?doid=224056.224070>).
- [59] Jim Gray, *The Transaction Concept: Virtues and Limitations*, w: „7th International Conference on Very Large Data Bases” (VLDB), wrzesień 1981 (<http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf>).
- [60] Hector Garcia-Molina i Kenneth Salem, *Sagas*, w: „ACM International Conference on Management of Data” (SIGMOD), maj 1987 (<http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>; <https://dl.acm.org/citation.cfm?doid=38713.38742>).
- [61] Pat Helland, *Memories, Guesses, and Apologies*, *blogs.msdn.com*, 15 maja 2007 (<https://blogs.msdn.microsoft.com/pathelland/2007/05/15/memories-guesses-and-apologies/>).
- [62] Yoongu Kim, Ross Daly, Jeremie Kim i in., *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*, w: „41st Annual International Symposium on Computer Architecture” (ISCA), czerwiec 2014 (<https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf>; <https://dl.acm.org/citation.cfm?doid=2678373.2665726>).
- [63] Mark Seaborn i Thomas Dullien, *Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges*, *googleprojectzero.blogspot.co.uk*, 9 marca 2015 (<https://googleprojectzero.blogspot.co.uk/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>).
- [64] Jim N. Gray i Catharine van Ingen, *Empirical Measurements of Disk Failure Rates and Error Rates*, Microsoft Research, MSR-TR-2005-166, grudzień 2005 (<https://www.microsoft.com/en-us/research/publication/empirical-measurements-of-disk-failure-rates-and-error-rates/>).

- [65] Annamalai Gurusami i Daniel Price, *Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021*, *bugs.mysql.com*, lipiec 2014 (<https://bugs.mysql.com/bug.php?id=73170>).
- [66] Gary Fredericks, *Postgres Serializability Bug*, *github.com*, wrzesień 2015 (<https://github.com/gfredericks/pg-serializability-bug>).
- [67] Xiao Chen, *HDFS DataNode Scanners and Disk Checker Explained*, *blog.cloudera.com*, 20 grudnia 2016 (<http://blog.cloudera.com/blog/2016/12/hdfs-datanode-scanners-and-disk-checker-explained/>).
- [68] Jay Kreps, *Getting Real About Distributed System Reliability*, *blog.empathybox.com*, 19 marca 2012 (<http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability>).
- [69] Martin Fowler, *The LMAX Architecture*, *martinfowler.com*, 12 lipca 2011 (<https://martinfowler.com/articles/lmax.html>).
- [70] Sam Stokes, *Move Fast with Confidence*, *blog.samstokes.co.uk*, 11 lipca 2016 (<http://blog.samstokes.co.uk/blog/2016/07/11/move-fast-with-confidence/>).
- [71] *Sawtooth Lake Documentation*, Intel Corporation, *intelledger.github.io*, 2016 (<https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html>).
- [72] Richard Gendal Brown, *Introducing R3 Corda™: A Distributed Ledger Designed for Financial Services*, *gendal.me*, 5 kwietnia 2016 (<https://gendal.me/2016/04/05/introducing-r3-corda-a-distributed-ledger-designed-for-financial-services/>).
- [73] Trent McConaghy, Rodolphe Marques, Andreas Müller i in., *BigchainDB: A Scalable Blockchain Database*, *bigchaindb.com*, 8 czerwca 2016 (<https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf>).
- [74] Ralph C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*, w: CRYPTO '87, sierpień 1987 (<https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf>; [https://link.springer.com/chapter/10.1007%2F3-540-48184-2\\_32](https://link.springer.com/chapter/10.1007%2F3-540-48184-2_32)).
- [75] Ben Laurie, *Certificate Transparency*, „ACM Queue”, rocznik 12, nr 8, s. 10 – 19, sierpień 2014 (<http://queue.acm.org/detail.cfm?id=2668154>; <https://dl.acm.org/citation.cfm?doid=2668152.2668154>).
- [76] Mark D. Ryan, *Enhanced Certificate Transparency and End-to-End Encrypted Mail*, w: „Network and Distributed System Security Symposium” (NDSS), luty 2014 (<https://eprint.iacr.org/2013/595.pdf>; <http://www.ndss-symposium.org/ndss2014/programme/enhanced-certificate-transparency-and-end-end-encrypted-mail/>).
- [77] *Software Engineering Code of Ethics and Professional Practice*, Association for Computing Machinery, *acm.org*, 1999 (<https://www.acm.org/about-acm>).
- [78] François Chollet, *Software development is starting to involve important ethical choices*, *twitter.com*, 30 października 2016 (<https://twitter.com/fchollet/status/792958695722201088>).
- [79] Igor Perisic, *Making Hard Choices: The Quest for Ethics in Machine Learning*, *engineering.linkedin.com*, listopad 2016 (<https://engineering.linkedin.com/blog/2016/11/making-hard-choices--the-quest-for-ethics-in-machine-learning>).

- [80] John Naughton, *Algorithm Writers Need a Code of Conduct*, *theguardian.com*, 6 grudnia 2015 (<https://www.theguardian.com/commentisfree/2015/dec/06/algorithm-writers-should-have-code-of-conduct>).
- [81] Logan Kugler, *What Happens When Big Data Blunders?*, „Communications of the ACM”, rocznik 59, nr 6, s. 15 – 16, czerwiec 2016 (<https://cacm.acm.org/magazines/2016/6/202655-what-happens-when-big-data-blunders/abstract>; <https://dl.acm.org/citation.cfm?doid=2942427.2911975>).
- [82] Bill Davidow, *Welcome to Algorithmic Prison*, *theatlantic.com*, 20 lutego 2014 (<https://www.theatlantic.com/technology/archive/2014/02/welcome-to-algorithmic-prison/283985/>).
- [83] Don Peck, *They're Watching You at Work*, *theatlantic.com*, grudzień 2013 (<https://www.theatlantic.com/magazine/archive/2013/12/theyre-watching-you-at-work/354681/>).
- [84] Leigh Alexander, *Is an Algorithm Any Less Racist Than a Human?*, *theguardian.com*, 3 sierpnia 2016 (<https://www.theguardian.com/technology/2016/aug/03/algorithm-racist-human-employers-work>).
- [85] Jesse Emspak, *How a Machine Learns Prejudice*, *scientificamerican.com*, 29 grudnia 2016 (<https://www.scientificamerican.com/article/how-a-machine-learns-prejudice/>).
- [86] Maciej Cegłowski, *The Moral Economy of Tech*, *idlewords.com*, czerwiec 2016 ([http://idlewords.com/talks/sase\\_panel.htm](http://idlewords.com/talks/sase_panel.htm)).
- [87] Cathy O'Neil, *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*, Crown Publishing, 2016, ISBN: 978-0-553-41881-1 (<https://weaponsofmathdestructionbook.com/>).
- [88] Julia Angwin, *Make Algorithms Accountable*, *nytimes.com*, 1 sierpnia 2016 (<https://www.nytimes.com/2016/08/01/opinion/make-algorithms-accountable.html>).
- [89] Bryce Goodman i Seth Flaxman, *European Union Regulations on Algorithmic Decision-Making and a „Right to Explanation”*, *arXiv:1606.08813*, 31 sierpnia 2016 (<https://arxiv.org/abs/1606.08813>).
- [90] *A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes*, raport, „United States Senate Committee on Commerce, Science, and Transportation”, *commerce.senate.gov*, grudzień 2013 (<https://www.commerce.senate.gov/public/index.cfm/reports?ID=57C428EC-8F20-44EE-BFB8-A570E9BE0CCC>).
- [91] Olivia Solon, *Facebook's Failure: Did Fake News and Polarized Politics Get Trump Elected?*, *theguardian.com*, 10 listopada 2016 (<https://www.theguardian.com/technology/2016/nov/10/facebook-fake-news-election-conspiracy-theories>).
- [92] Donella H. Meadows i Diana Wright, *Thinking in Systems: A Primer*, Chelsea Green Publishing, 2008, ISBN: 978-1-603-58055-7.
- [93] Daniel J. Bernstein, *Listening to a „big data”/„data science” talk*, *twitter.com*, 12 maja 2015 (<https://twitter.com/hashbreaker/status/598076230437568512>).
- [94] Marc Andreessen, *Why Software Is Eating the World*, „The Wall Street Journal”, 20 sierpnia 2011 (<https://genius.com/Marc-andreessen-why-software-is-eating-the-world-annotated>).

- [95] J.M. Porup, „Internet of Things” Security Is Hilariously Broken and Getting Worse, *arstechnica.com*, 23 stycznia 2016 (<https://arstechnica.com/information-technology/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/>).
- [96] Bruce Schneier, *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*, W.W. Norton, 2015, ISBN: 978-0-393-35217-7 ([https://www.schneier.com/books/data\\_and\\_goliath/](https://www.schneier.com/books/data_and_goliath/)).
- [97] The Grugq, *Nothing to Hide*, *grugq.tumblr.com*, 15 kwietnia 2016 (<https://grugq.tumblr.com/post/142799983558/nothing-to-hide>).
- [98] Tony Beltramelli, *Deep-Spying: Spying Using Smartwatch and Deep Learning*, praca magisterska, IT University of Copenhagen, grudzień 2015 (<https://arxiv.org/abs/1512.05616>).
- [99] Shoshana Zuboff, *Big Other: Surveillance Capitalism and the Prospects of an Information Civilization*, „Journal of Information Technology”, rocznik 30, nr 1, s. 75 – 89, kwiecień 2015 ([https://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=2594754](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2594754); <https://link.springer.com/article/10.1057%2Fjit.2015.5>).
- [100] Carina C. Zona, *Consequences of an Insightful Algorithm*, w: „GOTO Berlin”, listopad 2016 (<https://www.youtube.com/watch?v=YRI40A4tyWU>).
- [101] Bruce Schneier, *Data Is a Toxic Asset, So Why Not Throw It Out?*, *schneier.com*, 1 marca 2016 ([https://www.schneier.com/essays/archives/2016/03/data\\_is\\_a\\_toxic\\_asse.html](https://www.schneier.com/essays/archives/2016/03/data_is_a_toxic_asse.html)).
- [102] John E. Dunn, *The UK’s 15 Most Infamous Data Breaches*, *techworld.com*, 18 listopada 2016 (<https://www.techworld.com/security/uks-most-infamous-data-breaches-3604586/>).
- [103] Cory Scott, *Data is not toxic — which implies no benefit — but rather hazardous material, where we must balance need vs. want*, *twitter.com*, 6 marca 2016 ([https://twitter.com/cory\\_scott/status/706586399483437056](https://twitter.com/cory_scott/status/706586399483437056)).
- [104] Bruce Schneier, *Mission Creep: When Everything Is Terrorism*, *schneier.com*, 16 lipca 2013 ([https://www.schneier.com/essays/archives/2013/07/mission\\_creep\\_when\\_e.html](https://www.schneier.com/essays/archives/2013/07/mission_creep_when_e.html)).
- [105] Lena Ulbricht i Maximilian von Grafenstein, *Big Data: Big Power Shifts?*, „Internet Policy Review”, rocznik 5, nr 1, marzec 2016 (<https://policyreview.info/articles/analysis/big-data-big-power-shifts>).
- [106] Ellen P. Goodman i Julia Powles, *Facebook and Google: Most Powerful and Secretive Empires We’ve Ever Known*, *theguardian.com*, 28 września 2016 (<https://www.theguardian.com/technology/2016/sep/28/google-facebook-powerful-secretive-empire-transparency>).
- [107] *Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data*, Official Journal of the European Communities No. L 281/31, *eur-lex.europa.eu*, listopad 1995 (<http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:31995L0046>).
- [108] Brendan Van Alsenoy, *Regulating Data Protection: The Allocation of Responsibility and Risk Among Actors Involved in Personal Data Processing*, rozprawa doktorska, KU Leuven Centre for IT and IP Law, sierpień 2016 (<https://lirias.kuleuven.be/handle/123456789/545027>).

- [109] Michiel Rhoen, *Beyond Consent: Improving Data Protection Through Consumer Protection Law*, „Internet Policy Review”, rocznik 5, nr 1, marzec 2016 (<https://policyreview.info/articles/analysis/beyond-consent-improving-data-protection-through-consumer-protection-law>).
- [110] Jessica Leber, *Your Data Footprint Is Affecting Your Life in Ways You Can't Even Imagine*, *fastcoexist.com*, 15 marca 2016 (<https://www.fastcompany.com/3057514/your-data-footprint-is-affecting-your-life-in-ways-you-cant-even-imagine>).
- [111] Maciej Cegłowski, *Haunted by Data*, *idlewords.com*, październik 2015 ([http://idlewords.com/talks/haunted\\_by\\_data.htm](http://idlewords.com/talks/haunted_by_data.htm)).
- [112] Sam Thielman, *You Are Not What You Read: Librarians Purge User Data to Protect Privacy*, *theguardian.com*, 13 stycznia 2016 (<https://www.theguardian.com/us-news/2016/jan/13/us-library-records-purged-data-privacy>).
- [113] Conor Friedersdorf, *Edward Snowden's Other Motive for Leaking*, *theatlantic.com*, 13 maja 2014 (<https://www.theatlantic.com/politics/archive/2014/05/edward-snowdens-other-motive-for-leaking/370068/>).
- [114] Phillip Rogaway, *The Moral Character of Cryptographic Work*, *Cryptology ePrint* 2015/1162, grudzień 2015 (<http://web.cs.ucdavis.edu/~rogaway/papers/moral-fn.pdf>).





Warto zauważyć, że definicje w tym słowniczku są krótkie i proste. Mają one przekazywać podstawowe znaczenie pojęć, a nie ich wszystkie subtelne aspekty. Więcej informacji znajdziesz we wskazanych tu miejscach książki.

## Asymetria

1. Niezrównoważone obciążenie partycji (niektóre partycje otrzymują dużo żądań lub danych, a inne znacznie mniej). Stosowane jest też określenie *hot spot*. Zob. punkty „Asymetryczne obciążenie robocze i odciażanie hot spotów” i „Radzenie sobie z asymetrią”.
2. Anomalia związana z koordynacją czasu i powodująca wrażenie, że zdarzenia zachodzą w nieoczekiwanej, niesekwencyjnej kolejności. Zob. omówienie *odczytu zniekształconego* w punkcie „Izolacja snapshotów i powtarzalny odczyt”, *zapisu zniekształcającego* w punkcie „Zapis zniekształcający i fantomy” oraz *przesunięcia zegara* w punkcie „Używanie znaczników czasu do porządkowania zdarzeń”.

## Asynchroniczne

Bez oczekiwania na ukończenie jakiegoś zadania (np. przesłania danych w sieci do innego węzła) i bez przyjmowania założeń co do długości jego wykonywania. Zob. punkty „Replikacja synchroniczna i asynchroniczna”, „Sieci synchroniczne i asynchroniczne” i „Model systemu a rzeczywistość”.

## Atomowe

1. W kontekście współbieżności opisuje operację, która wygląda tak, jakby była wykonywana w jednym momencie. Dlatego inny jednoczesny proces nigdy nie widzi danej operacji jako „połowicznie ukończonej”. Zob. też *izolacja*.
2. W kontekście transakcji atomowość oznacza grupowanie zbioru zapisów, które trzeba albo razem zatwierdzić, albo wycofać (nawet po wystąpieniu błędu). Zob. punkty „Atomowość” i „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”.

## Bez zasobów współdzielonych

Architektura, w której niezależne węzły (każdy z własnymi procesorami, pamięcią i dyskami) są połączone tradycyjną siecią, a nie za pomocą pamięci lub dysków współużytkowanych. Zob. wprowadzenie do części II.

## Błędy bizantyjskie

Problem polegający na tym, że węzeł działa nieprawidłowo w nieoczekiwany sposób, np. wysyła sprzeczne lub szkodliwe komunikaty do innych węzłów. Zob. punkt „Błędy bizantyjskie”.

## Blokada

Mechanizm zapewniania, że tylko jeden wątek, węzeł lub transakcja mają dostęp do danego obiektu. Gdy inna jednostka chce uzyskać dostęp do tego samego obiektu, musi czekać na zwolnienie blokady. Zob. punkty „Blokady dwuetapowe” i „Lider i blokady”.

## Blokady dwuetapowe

Algorytm zapewniania izolacji na poziomie sekwencyjności. Działa w ten sposób, że transakcja zajmuje blokady dotyczące wszystkich wczytywanych lub zapisywanych danych i utrzymuje tę blokadę do momentu zakończenia transakcji. Zob. punkt „Blokady dwuetapowe”.

## Dane pochodne

Zbiór danych tworzony na podstawie innych danych za pomocą powtarzalnego procesu, który w razie potrzeby można uruchomić ponownie. Dane pochodne zwykle są potrzebne do przyspieszenia określonego rodzaju odczytów danych. Przykładowe dane pochodne to indeksy, pamięć podręczna i widoki zmaterializowane. Zob. wprowadzenie do części III.

## Deklaratywne

Opisuje cechy, jakie coś powinno mieć, ale bez precyzyjnego określania kroków do osiągnięcia celu. W kontekście zapytań optymalizator przyjmuje zapytanie deklaratywne i decyduje o najlepszym sposobie wykonania go. Zob. punkt „Język zapytań o dane”.

## Denormalizacja

Wprowadzanie nadmiarowości lub powtórzeń w *znormalizowanym* zbiorze danych (zwykle w formie *pamięci podręcznej* lub *indeksu*) w celu przyspieszenia odczytów. Wartość zdenormalizowana jest czymś w rodzaju wstępnie obliczonego wyniku zapytania i przypomina widok zmaterializowany. Zob. punkty „Operacje na pojedynczych obiektach i na wielu obiektach” i „Generowanie kilku widoków na podstawie tego samego dziennika zdarzeń”.

## Deterministyczne

Opisuje funkcję, która na podstawie tych samych danych wejściowych zawsze generuje identyczne dane wyjściowe. To oznacza, że funkcja nie może zależeć od liczb losowych, pory dnia, komunikacji sieciowej i innych nieprzewidywalnych czynników.

## Dziennik

Plik do składowania danych umożliwiający tylko dołączanie informacji. *Dziennik z zapisem wyprzedzającym* służy do zapewniania odporności systemu składowania danych na awarie (zob. punkt „Zapewnianie niezawodności b-drzew”). System składowania *o strukturze dziennika* używa dzienników jako podstawowego formatu przechowywania danych (zob. punkt „Pliki SSTable i drzewa LSM”). *Dziennik replikacji* służy do kopiowania zapisów z lidera do obserwatorów (zob. punkt „Liderzy i obserwatorzy”). *Dziennik zdarzeń* może reprezentować strumień danych (zob. punkt „Podział dzienników na partycje”).

## ETL

Od ang. *extract-transform-load*. Proces pobierania danych z bazy źródłowej, przekształcania ich na postać bardziej dostosowaną do zapytań analitycznych i wczytywania do hurtowni danych lub systemu przetwarzania wsadowego. Zob. punkt „Hurtownie danych”.

## Graf

Struktura danych składająca się z *wierzchołków* (rzeczy, do których możesz się odnosić; inna nazwa to *węzły*) i *krawędzi* (połączeń między wierzchołkami; inne określenia to *relacje* lub *łuki*). Zob. punkt „Modele danych przypominające graf”.

## Hurtownia danych

Baza danych, w której informacje z kilku różnych systemów OLTP zostały połączone i przygotowane na potrzeby analityki. Zob. punkt „Hurtownia danych”.

## Idempotentny

Opisuje operację, którą można bezpiecznie ponowić. Jeśli taka operacja jest wykonywana kilkakrotnie, efekt zawsze jest taki sam jak po jej jednokrotnym wykonaniu. Zob. punkt „Idempotencja”.

## Indeks

Struktura danych umożliwiająca wydajne wyszukiwanie wszystkich rekordów o ustalonej wartości określonego pola. Zob. punkt „Struktury danych używane w bazie”.

## Indeks pomocniczy

Dodatkowa struktura danych utrzymywana obok głównego magazynu danych i umożliwiająca wydajne wyszukiwanie rekordów spełniających określone warunki. Zob. punkty „Inne struktury używane dla indeksów” i „Podział na partycje a indeksy pomocnicze”.

## **Izolacja**

W kontekście transakcji opisuje, do jakiego stopnia jednocześnie wykonywane transakcje mogą wzajemnie zakłócać swoją pracę. Poziom *sekwencyjności* zapewnia największe gwarancje, jednak stosowane są też niższe poziomy izolacji. Zob. punkt „Izolacja”.

## **Klucz główny**

Wartość (zwykle liczba lub łańcuch znaków), która jednoznacznie identyfikuje rekord. W wielu aplikacjach klucze główne są generowane przez system (sekwencyjnie lub losowo) w momencie tworzenia rekordu. Te klucze zwykle nie są ustawiane przez użytkowników. Zob. też *indeks pomocniczy*.

## **Konsensus**

Fundamentalny problem w przetwarzaniu rozproszonym. Dotyczy osiągania zgody kilku węzłów w jakiejś kwestii (np. tego, który węzeł powinien być liderem w klastrze bazodanowym). Ten problem jest znacznie trudniejszy, niż się początkowo wydaje. Zob. punkt „Konsensus z zachowaniem odporności na błędy”.

## **Kontrola przepływu**

Zob. *wsteczna propagacja*.

## **Kworum**

Minimalna liczba węzłów, które muszą zagłosować na operację, aby została uznana za zakończoną powodzeniem. Zob. punkt „Odczyty i zapisy zgodne z kworum”.

## **Lider**

Gdy dane lub usługa są replikowane w kilku węzłach, liderem jest wyznaczona replika, która może wprowadzać zmiany. Lider może być wybierany za pomocą jakiegoś protokołu lub wskazywany ręcznie przez administratora. Inne nazwy to *replika główna* lub *nadrzędna*. Zob. punkt „Liderzy i obserwatorzy”.

## **Liniowe**

Działanie w taki sposób, jakby w systemie znajdowała się tylko jedna kopia danych. Zob. punkt „Liniowość”.

## **Lokalność**

Optymalizacja wydajności polegająca na umieszczaniu kilku fragmentów danych w tym samym miejscu, jeśli są często potrzebne w tym samym czasie. Zob. punkt „Lokalność danych w kontekście zapytań”.

## Materializowanie

Wykonywanie obliczeń wcześniej i zapisywanie ich wyników (w odróżnieniu od przeprowadzania obliczeń na żądanie). Zob. punkty „Agregacja — kostki danych i widoki zmateralizowane” i „Materializowanie stanu pośredniego”.

## Nieograniczone

Bez znanego górnego ograniczenia lub rozmiaru. Odwrotność określenia *ograniczone*.

## Obserwator

Replika, która nie akceptuje bezpośrednio zapisów od klientów, a jedynie przetwarza zmiany w danych otrzymane od lidera. Inne nazwy to: *replika pomocnicza*, *replika podrzędna*, *replika przeznaczona do odczytu* lub *rezerwa dynamiczna*. Zob. punkt „Liderzy i obserwatorzy”.

## Odporność na błędy

Zdolność do automatycznego przywracania stanu po wystąpieniu problemów (np. po awarii maszyny lub zerwaniu połączenia sieciowego). Zob. punkt „Niezawodność”.

## Ograniczone

Mające znaną górną granicę lub rozmiar. Używane np. w kontekście opóźnienia sieciowego (zob. punkt „Limity czasu i nieograniczone opóźnienia”) i zbiorów danych (zob. wprowadzenie do rozdziału 11.).

## OLAP

Od ang. *online analytic processing*. Wzorzec dostępu charakteryzujący się agregowaniem (np. zliczaniem, sumowaniem, uśrednianiem) dużej liczby rekordów. Zob. punkt „Przetwarzanie transakcji czy analityka?”.

## OLTP

Od ang. *online transaction processing*. Wzorzec dostępu charakteryzujący się szybko wykonywanymi zapytaniami, które wczytują lub zapisują niewielką liczbę rekordów, zwykle indeksowanych za pomocą klucza. Zob. punkt „Przetwarzanie transakcji czy analityka?”.

## Pamięć podręczna

Komponent zapamiętujący niedawno używane dane, aby przyspieszyć ich późniejsze odczyty. Zwykle nie jest kompletna. Jeśli jakichś danych nie ma w pamięci podręcznej, trzeba je pobrać z podstawowego, wolniejszego systemu składowania danych zawierającego ich kompletną kopię.

## Percentyle

Sposób pomiaru rozkładu wartości w wyniku zliczania, ile wartości znajduje się powyżej lub poniżej jakiegoś progu. Na przykład percentyl 95% dla czasów odpowiedzi z jakiegoś okresu to taki czas  $t$ , że 95% żądań z tego okresu jest wykonywanych szybciej niż w czasie  $t$ , a 5% w czasie dłuższym niż  $t$ . Zob. punkt „Opis wydajności”.

## Podział na partycje

Podział dużych zbiorów danych lub obliczeń, które są zbyt wymagające dla jednej maszyny, na mniejsze porcje i rozdzielanie ich między kilka maszyn (ang. *partitioning* lub *sharding*). Zob. rozdział 6.

## Procedura składowana

Taki sposób zapisu logiki transakcji, że może być ona w całości wykonywana na serwerze bazodanowym bez komunikowania się z klientem. Zob. punkt „Rzeczywiste sekwencyjne wykonywanie transakcji”.

## Proces strumieniowy

Stale wykonywane obliczenia, które pobierają niekończący się strumień zdarzeń jako dane wejściowe i na tej podstawie generują dane wyjściowe. Zob. rozdział 11.

## Proces wsadowy

Obliczenia przyjmujące ustalony (i zwykle duży) zbiór danych wejściowych i generujący dane wyjściowe bez modyfikowania wejścia. Zob. rozdział 10.

## Przekroczenie limitu czasu

Jeden z najprostszych sposobów wykrywania błędów, który polega na zaobserwowaniu braku odpowiedzi w określonym czasie. Nie da się jednak ustalić, czy przekroczenie limitu czasu wynika z kłopotów ze zdalnym węzłem, czy z problemów z siecią. Zob. punkt „Limity czasu i nieograniczone opóźnienia”.

## Przełączanie awaryjne

W systemach z jednym liderem przełączanie awaryjne to proces przekazywania roli lidera innemu węzłowi. Zob. punkt „Radzenie sobie z przestojami węzłów”.

## Przyczynowość

Zależność między zdarzeniami powstająca, gdy jedna rzecz „zdarzyła się wcześniej” niż inna w systemie. Na przykład późniejsze zdarzenie może być odpowiedzią na wcześniejsze, być oparte na starszym lub być niezrozumiałe bez niego. Zob. punkty „Relacja »zdarzyło się wcześniej« i równoległość” i „Uporządkowanie i przyczynowość”.

## Replikacja

Przechowywanie kopii tych samych danych w kilku węzłach (*replikach*), dzięki czemu pozostają one dostępne, gdy do któregoś z węzłów nie można dotrzeć. Zob. rozdział 5.

## Równoważenie

Przenoszenie danych lub usług z jednego węzła na inny w celu równomiernego rozłożenia obciążenia. Zob. punkt „Równoważenie partycji”.

## Rozproszone

Działające w kilku węzłach połączonych siecią. Charakteryzuje się *awariami częściowymi* — część systemu może być uszkodzona, podczas gdy pozostałe wciąż działają. Często nie da się wtedy precyzyjnie stwierdzić, co nie funkcjonuje. Zob. punkt „Błędy i awarie częściowe”.

## Rozszczepiony mózg

Scenariusz, w którym dwa węzły jednocześnie uważają, że są liderem. Może to powodować naruszenie gwarancji w systemie. Zob. punkty „Radzenie sobie z przestojami węzłów” i „Prawda jest definiowana przez większość”.

## Schemat

Opis struktury danych, w tym jej pól i typów danych. Zgodność danych ze schematem można sprawdzać w różnych momentach ich życia (zob. punkt „Elastyczność schematu w modelu opartym na dokumentach”). Z czasem schemat może się zmieniać (zob. rozdział 4.).

## Sekwencyjność

Gwarancja, że jeśli kilka transakcji jest wykonywanych jednocześnie, działają one tak, jakby były przeprowadzane jedna po drugiej w kolejności sekwencyjnej. Zob. punkt „Sekwencyjność”.

## Skrót

Wynik funkcji przekształcającej dane wejściowe w liczbę wyglądającą na losową. Te same dane wejściowe zawsze dają tę samą liczbę wyjściową. Dla różnych danych wejściowych zwykle otrzymywane są inne dane wyjściowe (choć zdarza się, że są one takie same; jest to tzw. *kolizja*). Zob. punkt „Podział na podstawie skrótów kluczy”.

## Synchroniczne

Odwrotność określenia *asynchroniczne*.

## System zapisu

System przechowujący główną, miarodajną wersję jakichś danych. Można go nazwać *źródłem prawdy*. Zmiany są najpierw zapisywane w tym miejscu, a następnie na podstawie systemu zapisu można generować inne zbiory danych. Zob. wprowadzenie do części III.

## Transakcja

Grupa kilku odczytów i zapisów połączona w logiczną jednostkę w celu uproszczenia obsługi błędów i problemów ze współbieżnością. Zob. rozdział 7.

## Trwałe

Dane przechowywane w taki sposób, że wierzysz, iż nie zostaną utracone nawet po wystąpieniu różnych błędów. Zob. punkt „Trwałość”.

## Twierdzenie CAP

Powszechnie błędnie rozumiane twierdzenie teoretyczne, nieprzydatne w praktyce. Zob. punkt „Twierdzenie CAP”.

## Uporządkowanie całkowite

Sposób porównywania elementów (np. znaczników czasu) pozwalający zawsze stwierdzić, który z dwóch obiektów jest większy, a który mniejszy. Jeśli niektórych elementów nie da się ze sobą porównać (nie można stwierdzić, który jest większy lub mniejszy), występuje *uporządkowanie częściowe*. Zob. punkt „Uporządkowanie przyczynowe nie jest uporządkowaniem całkowitym”.

## Węzeł

Instancja oprogramowania działająca na komputerze i komunikująca się z innymi węzłami za pomocą sieci w celu wykonania jakiegoś zadania.

## Wsteczna propagacja

Zmuszanie nadawcy danych do spowolnienia, ponieważ odbiorca nie nadąża z przetwarzaniem. Inna nazwa to kontrola przepływu. Zob. punkt „Systemy obsługi komunikatów”.

## Wyszukiwanie pełnotekstowe

Przeszukiwanie tekstu na podstawie dowolnych słów kluczowych, często z wykorzystaniem dodatkowych mechanizmów takich jak dopasowywanie słów o podobnej pisowni lub synonimów. Indeks pełnotekstowy jest *indeksem pomocniczym* wspomagającym wykonywanie takich zapytań. Zob. punkt „Wyszukiwanie pełnotekstowe i indeksy rozmyte”.

## Zatwierdzanie dwuetapowe

Algorytm gwarantujący, że cała grupa węzłów bazodanowych albo zatwierdzi, albo anuluje transakcję. Zob. punkt „Zatwierdzanie atomowe i zatwierdzanie dwuetapowe”.



## **Złączenie**

Łączenie rekordów mających ze sobą coś wspólnego. Najczęściej stosowane w sytuacji, gdy jeden rekord obejmuje referencję do innego (klucz obcy, referencję do dokumentu lub krawędź w grafie), a w zapytaniu potrzebny jest rekord, do którego ta referencja prowadzi. Zob. punkty „Relacje wiele do jednego i wiele do wielu” oraz „Złączanie i grupowanie po stronie reducera”.

## **Znormalizowane**

Ustrukturyzowane w taki sposób, że nie występują nadmiarowość ani duplikaty. W znormalizowanej bazie po modyfikacji konkretnych danych wystarczy zmienić je w jednym miejscu. Nie trzeba modyfikować wielu kopii w różnych lokalizacjach. Zob. punkt „Relacje wiele do jednego i wiele do wielu”.

## A

- ACID, 225
- agregacje zmaterializowane, 108
- agregowanie
  - danych, 108
  - w pamięci, 383
- aktor, 143
- aktualizowanie, 241, 243
  - danych, 136
  - stanu pochodnego, 479
  - widoków zmaterializowanych, 450
- aktualność, 505
- aktualny stan systemu, 442
- aktywność, 303
  - użytkownika, 392
- algorytmy
  - osiągania konsensusu, 355
    - ograniczenia, 357
  - rozwiązywania konfliktów, 478
  - SSI, 258
- analityka, 98
  - prognostyczna, 513
  - schematy, 101
- analiza
  - dziennika, 381
  - przepływów danych, 475
  - strumieni, 449
  - zdarzeń, 392
- Apache Avro, 128
- Apache Thrift, 124
- API, 413, 440
- architektura
  - bez zasobów współdzielonych, 152
  - lambda, 480
  - SIMD, 106
- asymetria, 395
- asymetryczne obciążenie robocze, 206
- asynchroniczne wykrywanie konfliktów, 175
- atomowe
  - operacje zapisu, 242
  - zatwierdzanie transakcji, 343, 362
- atomowość, 225, 228
- audyty, 511, 512
- automatyczne
  - rozwiązywanie konfliktów, 178
  - wykrywanie utraconych aktualizacji, 243
- Avro, 128
- awaria, 84, 302, 359, 461
  - koordynatora, 348, 352
  - lidera, 162
  - obserwatora, 162
- awarie częściowe, 272

## B

- baza danych DynamoDB, 180
- bazy danych
  - federacyjne, 484
  - kolumnowe, 103
    - agregowanie, 108
    - kompresja kolumn, 105
    - sortowanie, 107
    - zapis, 108
  - MPP, 402, 404
  - OLTP, 101
  - oparte na dokumentach, 49, 51
  - oparte na grafach, 69
  - relacyjne, 51
  - z podziałem na komponenty, 484
- BBS, bulletin board system, 384

b-drzewo, 89  
  optymalizacja, 91  
  zapewnianie niezawodności, 91  
bezpieczeństwo, 303  
biblioteka  
  Protocol Buffers, 124  
  Thrift, 124  
blokady, 243, 296, 323, 362  
  dwuetapowe, 254, 264  
    implementowanie, 255  
    wydajność, 256  
  oparte na predyktach, 256  
  zakresów indeksu, 257  
bloki, 89  
błędy, 272, 404, 412, 459  
  bizantyjskie, 298, 302  
  ludzkie, 24  
  programowe, 23, 509  
  sieci, 276  
  sprzętowe, 23  
  typu awaria  
    przerwanie pracy, 302  
    przywrócenie stanu, 302  
brokery komunikatów, 142, 429  
  oparty na dziennikach, 462  
  w modelu AMQP, 462  
brudne  
  odczyty, 234, 263  
  zapisy, 235, 263  
BSP, bulk synchronous parallel, 411

## C

CEP, complex event processing, 449  
chmura, 273  
CQRS, command query responsibility segregation, 445  
Cypher, 63  
czas  
  wystąpienia zdarzenia, 452  
  zwracania odpowiedzi, 293  
członkostwo, 358

## D

dane  
  archiwalne, 136  
  jako  
    zasoby, 520  
    źródło władzy, 520  
  pochodne, 375, 475  
  rozproszone, 151  
  typu klucz-wartość, 203  
Datalog, 69  
DDD, domain-driven design, 441  
drzewa zrównoważone, 87  
drzewo LSM, 85, 87, 92  
duplikaty, 499  
dynamiczne określanie typów, 133  
Dynamo, 180  
dysk SSD, 84, 97  
dziennik, 80, 82  
  do przechowywania komunikatów, 432  
  logiczny, 165  
  par klucz-wartość, 83  
  replikacji, 163  
  WAL, 91, 164  
  zapisu z wyprzedzeniem, 91  
  zdarzeń, 442, 445  
  z obsługą komunikatów, 433  
dzierżawy, 362

## E

eksploatacja, 33, 213  
elastyczność schematu, 52  
eliminowanie entropii, 181  
entropia, 181  
ETL, Extract-Transform-Load, 100  
event sourcing, 441  
ewolucja schematów, 127  
  reguły, 131

## F

fantomy, 245, 248, 264  
federacyjne bazy danych, 484

- filozofia Uniksa, 383
  - eksperymenty, 386
  - jednolity interfejs, 384
  - przezroczystość, 386

#### format

- Avro, 128
- BinaryProtocol, 125
- JSOM, 121
- SSTable, 85
- XML, 121

#### formaty

- przechowywania danych, 80
- kodowania danych, 120

## G

#### generowanie

- danych, 474
- kodu, 133
- numerów porządkowych, 336
- widoków, 445

GFS, Google File System, 387

GiST, Generalized Search Tree, 96

graf, 60, 410

- właściwości, 61

- zależności przyczynowych, 191

grupowanie, 391

#### gwarancje

- czasu zwracania odpowiedzi, 293
- spójności, 316
- unikatowości, 323
- uporządkowania, 331, 339

## H

Hadoop, 402

haszowanie, 81

- spójne, 205

hurtownie danych, 99, 101, 110

## I

idempotencja, 460

- operacji, 498

identyfikatory operacji, 500

#### implementowanie

- bazy liniowej, 340
- blokad dwuetapowych, 255
- dzienników replikacji, 163
- izolacji snapshotów, 238
- odczytu, 236
- przechwytywania zmian w danych, 438
- rozgłaszania z uporządkowaniem całkowitym, 342
- systemów liniowych, 325

#### indeksy, 81, 240

- dzielone na podstawie pojęć, 217
- dzielone według dokumentów, 217
- GiST, 96
- klastrowe, 95
- pokrywające, 95
- pomocnicze, 94, 207, 488
- rozmyte, 96
- wielokolumnowe, 95
- wyszukiwania, 399
- wyszukiwania pełnotekstowego, 488
- z dołączonymi kolumnami, 95

integralność, 505, 509

integrowanie danych, 474

interfejs API, 413, 440

inwigilacja, 517

izolacja, 226, 229

- snapshotów, 236, 238, 240

- transakcji, 233

## J

jednoczesność, 189

#### język

- Avro IDL, 128
- Cypher, 63
- Datalog, 69
- SPARQL, 66, 68
- WSDL, 138

#### języki

- deklaratywne zapytań, 413
- z dynamicznym określaniem typów, 133
- zapytań o dane, 54

JSOM, 121

## K

- klauzula GROUP BY, 394
- klienty stanowe, 493
  - przekazywanie zmian stanu, 494
  - tryb offline, 493
- klucz główny, 94
- kod aplikacji
  - do rozwiązywania konfliktów, 177
  - jako funkcja generująca, 488
  - oddzielony od stanu, 488
  - zmiany stanu, 489
- kodowanie danych, 120, 141
  - binarne, 122, 124
  - CompactProtocol, 126
- kolejki, 279
- kolumny, 105
- komponenty, 482
- kompresja
  - dziennika, 83, 440
  - kolumn, 105
  - segmentów, 83
- komunikaty, 142, 427
  - starsze, 436
- konflikty, 177
  - automatyczne rozwiązywanie, 178
  - niestandardowy kod, 177
  - unikanie, 175
  - wykrywanie, 175
- konsensus, 343, 354–356
- konserwacja, 33
  - aplikacji, 19
- konsumenci, 433, 435
- kontrola współbieżności, 84, 446
- konwergencja baz
  - opartych na dokumentach, 54
  - relacyjnych, 54
- koordynacja, 358, 508
- kopie zapasowe, 237
- korekcja błędów, 275
- kostka
  - danych, 109
  - OLAP, 109
- koszty liniowości, 327
- kwora, 182, 326, 356
  - niepełne, 185

## L

- lambda, 480
- lider, 158, 296
- limity czasu, 278
- liniowe operacje atomowe, 359
- liniowość, 317, 319, 322, 326
  - koszty, 327
  - opóźnienia sieciowe, 330
- logiczne znaczniki czasu, 478
- lokalność danych, 53
- LSM, Log-Structured Merge-Tree, 88, 92

## Ł

- łańcuch wywołań, 382
- łatwość
  - eksploatacji, 33
  - konserwacji, 33
  - modyfikowania, 36
- łączenie powiązanych danych, 394

## M

- MapReduce, 57, 386, 388, 406
  - dane wyjściowe, 399
  - przepływ pracy, 390, 398
  - rozproszone wykonywanie operacji, 389
  - wykonywanie zadań, 388
- materializowanie, 410
  - konfliktów, 249
  - stanu pośredniego, 406
- mechanizm hinted-handoff, 185
- memtable, 87
- metabaza, 484
- migracja schematów, 480
- mikroporcje, 459
- model
  - aktora, 143
  - asynchroniczny, 301
  - częściowo synchroniczny, 301
  - danych, 41
    - RDF, 67
  - typu graf, 60
- MapReduce, 57

- NoSQL, 43
- obiektowy, 43
- oparty na dokumentach, 42
- przetwarzania, 403
- relacyjny, 42, 50
- sieciowy, 49, 69
- synchroniczny, 301
- triplestore, 66
- modyfikowanie, 36
  - aplikacji, 479
- monitorowanie nieaktualnych danych, 185
- MPP, massively parallel processing, 402
- MVCC, 260

## N

- narzędzia unixowe, 380
- niemodyfikowalność, 443
  - ograniczenia, 446
- niezawodność, 22
  - b-drzew, 91
- numerowanie epok, 356

## O

- obciążenie, 26, 30
  - asymetryczne, 206
- obserwator, 158
  - tworzenie, 161
- obserwowanie stanu pochodnego, 491
- obsługa
  - błędów, 232
  - danych wyjściowych, 401
  - komunikatów, 427, 433
  - przestojów węzłów, 181
  - strumieni zmian, 440
- odciążanie hot spotów, 206
- odczyt, 495
  - brudny, 234
  - fantomów, 264
  - monotoniczny, 169
  - powtarzalny, 236, 241
  - własnych zapisów, 167
  - zatwierdzonych danych, 234
  - ze spójnym przedrostkiem, 170

- zegara, 289
- zniekształcony, 237
- odporność na błędy, 354, 409, 412, 416, 459
- odtworzenie stanu po awarii, 461
- odzworowywanie modeli systemu, 303
- odzyskiwanie
  - pamięci, 294
  - stanu po awarii koordynatora, 352
  - stanu z nadrabianiem zaległości, 162
- ograniczenia
  - niemodyfikowalności, 446
  - transakcji rozproszonych, 353
  - uporządkowania całkowitego, 476
- okno
  - przesuwne, 455
  - rozłączne, 455
  - sesyjne, 455
  - skokowe, 455
- określanie spójności, 184
- OLAP, online analytic processing, 99
- OLTP, online transaction processing, 98
- operacje na obiektach, 228
- opis
  - obciążenia, 26
  - wydajności, 28
- opóźnienia
  - nieograniczone, 278
  - sieciowe, 281, 330
  - zmienne, 283
  - replikacji, 166, 171
- optymalizacje
  - b-drzew, 91
  - wydajności, 88
- ORM, object-relational mapping, 232
- osiąganie konsensusu, 354, 355

## P

- pamięć
  - podręczna, 488, 492
  - RAM, 97
- para klucz-wartość, 82
- partycje, 201
  - przeciążone, 203
  - strategie równoważenia, 210

perspektywy punktów końcowych, 501  
 pętle sprzężenia zwrotnego, 515  
 platformy z rozproszonymi aktorami, 143  
 pliki  
   samoopisowe, 133  
   SSTable, 85, 86, 87  
 podział, 153  
   asymetryczny, 203  
   baz danych, 482  
   dzienników na partycje, 432  
   indeksów pomocniczych  
     na podstawie dokumentów, 207  
     według pojęć, 208  
   na komponenty, 485, 486, 487  
   na partycje, 201, 253  
     dane typu klucz-wartość, 203  
     dynamiczny, 212  
     indeksy pomocnicze, 207  
     proporcjonalnie do liczby węzłów, 213  
     przedziały kluczy, 203  
     skrótory kluczy, 204  
     stała liczba, 211  
     według przedziałów kluczy, 217  
     z użyciem skrótów, 217  
   sieci na partycje, 277  
 ponowne  
   dostarczanie, 431  
   przetwarzanie, 479, 498  
 poprawność systemów przepływu danych, 506  
 porządkowanie zdarzeń, 477  
 potwierdzenia, 431  
 powiadomienia o zmianach, 359  
 poziomy izolacji, 233  
 predykat, 256  
 procedury składowane, 251, 252  
 proces ETL, 100  
 procesy wsadowe, 399, 401  
 producenci, 433, 435  
 projektowanie aplikacji, 487  
 Protocol Buffers, 124  
 protokół  
   IP, 275  
   SOAP, 138  
   TCP, 275, 279  
   UDP, 280  
 prywatność, 516, 518  
 przechowywanie  
   dziennika, 82  
   komunikatów, 432  
   wartości w indeksie, 95  
 przechwytywanie zmian w danych, 438  
   implementowanie, 438  
 przeciążenie sieci, 279  
 przekazywanie komunikatów, 142, 451  
 przełączanie awaryjne, 162  
 przenoszenie danych  
   do hurtowni, 100  
   do pamięci, 106  
 przepływ  
   danych, 134, 487, 489  
     przekazywanie komunikatów, 142  
     z użyciem usług, 137  
     z wykorzystaniem baz, 135  
   pracy, 390  
 przestoje węzłów, 161, 181  
 przestrzeganie więzów, 502  
 przesyłanie  
   dziennika WAL, 164  
   komunikatów, 503  
   strumieni zdarzeń, 426  
 przeszukiwanie strumieni, 450  
 przetwarzanie  
   analityczne w czasie rzeczywistym, 99  
   danych  
     ponowne, 479  
     z wielu partycji, 496  
   iteracyjne, 410  
   komunikatów, 350  
   strumieniowe, 425, 447, 478  
     zastosowania, 448  
   transakcji, 98  
     w czasie rzeczywistym, 98  
   w chmurze, 273  
   wektorowe, 106  
   wsadowe, 379, 380, 478  
   złożonych zdarzeń, 449  
   zadań, 504  
 przyczynowość, 331, 477  
 przydział zadań do węzłów, 360  
 przywracanie stanu po awarii, 84

punkty  
kontrolne, 459  
końcowe, 495, 498

## R

RDD, resilient distributed dataset, 409  
referencje do dokumentów, 391  
reguły  
ewolucji schematów, 131  
widoczności, 240  
rekordy częściowo zapisane, 84  
relacja  
„zdarzyło się wcześniej”, 189, 190  
wiele do jednego, 46  
wiele do wielu, 46  
relacyjne bazy typu MPP, 216  
replika, 158  
nadrzędna, 158  
podrzędna, 158  
replikacja, 153, 157  
asynchroniczna, 159, 167  
bez lidera, 180, 194  
oparta na instrukcjach, 163  
oparta na liderze, 158  
oparta na wyzwalaczach, 165  
opóźnienie, 166, 171  
synchroniczna, 159  
z jednym liderem, 194, 356  
z użyciem dziennika logicznego, 165  
z wieloma liderami, 171, 194  
konflikt przy zapisie, 174  
przypadki użycia, 172  
topologie, 178  
REST, 137  
rezerwa dynamiczna, 158  
rodzaje okien, 454  
rozgałęzianie, 430  
rozgłaszanie z uporządkowaniem całkowitym, 339, 355, 362, 503  
implementowanie, 342  
implementowanie bazy liniowej, 340  
rozproszone  
bazy danych, 402  
systemy plików, 386, 402  
wykonywanie operacji, 389

równoległe wykonywanie zapytań, 216  
równoległość, 189  
równoważenie

automatyczne, 213  
obciążenia, 430  
partycji, 210  
ręczne, 213

RPC, 137  
kodowanie danych, 141  
modyfikacje, 141  
wywołania, 139, 140

## S

samoregulacja, 521  
scalanie segmentów, 83–86  
schemat, 133  
generowany dynamicznie, 132  
gwiazdy, 101, 102  
odczytu, 130  
płatka śniegu, 103  
zapisu, 130, 131  
segmenty, 89  
sekwencyjne wykonywanie transakcji, 250, 254  
sekwencyjność, 322  
sieci synchroniczne i asynchroniczne, 281  
sieć semantyczna, 67  
SIMD, single-instruction-multi-data, 106  
skalowalność, 26, 151  
skalowanie odczytów, 166  
składowanie danych  
archiwalnych, 136  
łączenie technologii, 483  
skróty kluczy, 204  
słabe formy klamania, 300  
słownik, 81  
snapshot, 236, 240, 439  
SOA, service-oriented architecture, 137  
sortowanie, 107, 383  
przez scalanie, 85, 393  
w bazach kolumnowych, 107  
według klucza, 85  
SPARQL, 66, 68



- spójność, 184, 226, 315
  - odczytu po zapisie, 167
  - odczytu własnych zapisów, 167
  - ostateczna, 166
  - przyczynowa, 334
- SQL
  - zapytania o grafy, 64
- SSI, serializable snapshot isolation, 258, 264
- SSTable, Sorted String Table, 85, 87
- stan, 443
  - pochodny, 491
- strony, 89
- struktura
  - danych, 79
    - b-drzewo, 89
    - drzewo LSM, 92
    - dziennik, 82
    - indeks, 81
    - memtable, 87
    - tablica z haszowaniem, 81
  - grafu, 61
- struktury drzewiaste, 87
- strumienie, 426, 436, 443
  - zdarzeń, 426, 495
- superkomputery, 273
- synchroniczne wykrywanie konfliktów, 175
- synchronizacja
  - systemów, 437
  - zegarów, 285
- system
  - analityczny, 99
  - BBS, 384
  - bez zasobów współużytkowanych, 274
  - Bitcask, 82
  - CEP, 449
  - danych, 20
    - pochodnych, 375
    - unikający koordynacji, 508
  - liniowy, 319
    - implementacja, 325
  - obietnic, 347
  - obliczeniowy o wysokiej wydajności, 273
  - obsługi komunikatów, 427
  - OLAP, 99
  - OLTP, 99, 110
  - plików, 386
    - GFS, 387
    - HDFS, 387
    - QFS, 387
  - Pregel, 411
  - przepływu danych, 407, 506
  - przetwarzania
    - strumieniowego, 380, 451, 490
    - wsadowego, 379
  - RDBMS, 42
  - rozproszony, 271
  - składowania danych
    - oparty na stronach, 89
    - struktura memtable, 87
    - ze strukturą dziennika, 80, 111
  - z podziałem na komponenty, 486
  - zapisu, 375
  - zintegrowany, 486

## Ś

śledzenie, 516

## T

- tabela faktów, 101
- tablica z haszowaniem, 81, 84
- technika MVCC, 260
- technologie składowania danych, 483
- Thrift, 124
- tokeny odgradzające, 297
- topologie replikacji z wieloma liderami, 178
- transakcje, 223
  - anulowania, 232
  - blokady dwuetapowe, 254
  - dla wielu obiektów, 231
  - gwarancje ACID, 225
  - izolacja, 233
  - komercyjne, 98
  - OLTP, 251
  - rozproszone, 343, 349, 475
    - ograniczenia, 353
  - sekwencje rzeczywiste, 251
  - sekwencyjność, 250

- w czasie rzeczywistym, 98
- w procedurach składowanych, 251
- XA, 351

trasowanie żądań, 214

trwałość, 227

twierdzenie CAP, 328

tworzenie

- drzew LSM, 87

- globalnych snapshotów, 290

- indeksów wyszukiwania, 399

- indeksu, 483

- obserwatorów, 161

- plików SSTable, 87

- systemów danych, 512

typy danych, 128

## U

uczenie maszynowe, 488

ujednolicanie przetwarzania wsadowego i

- strumieniowego, 481

unikatowość, 323, 503

uporządkowanie, 331

- całkowite, 333, 339

- ograniczenia, 476

- operacji, 359

- przyczynowe, 333

- według numerów porządkowych, 335

- znaczniki czasu, 338

usługa, 379, 490

- REST, 137

- RPC, 137

usługi

- internetowe, 138

- zarządzania członkostwem i koordynacją, 362

- związane z członkostwem, 361

uspójnianie przy odczycie, 181

ustalanie zakończenia okna, 453

ustawodawstwo, 521

usuwanie rekordów, 84

utrata aktualizacji, 241, 264

utrzymywanie

- blokad, 352

- synchronizacji systemów, 437

uzyskiwanie aktualnego stanu, 442

## W

WAL, write-ahead log, 91

wektory wersji, 192

weryfikowanie, 510

węzeł, 181

widoki zmaterializowane, 109, 450, 492

więzy, 323, 507

- integralności, 238

- unikatowości, 362, 502

wnioskowanie na temat czasu, 451

wsadowe przepływy pracy, 398

współbieżność, 84, 446

wstrzymywanie procesów, 291

wybór lidera, 323, 343

wydajność, 28, 88, 98, 256

- techniki SSI, 262

wykonywanie równoległe, 412

wykorzystanie danych, 518

wykrywanie

- awarii, 359

- błędów, 277

- jednoczesnych zapisów, 187

- konfliktów, 175

- odczytu nieaktualnych wersji, 260

- usług, 360

- utraconych aktualizacji, 243

- zapisów, 261

wymuszanie przestrzegania więzów, 502

wyszukiwanie pełnotekstowe, 96

wywołania RPC, 139, 140, 451

wyzwalacz, 165

względność, 189

## X

XML, 121

## Z

zadania w modelu MapReduce, 388

zależności czasowe

- między kanałami, 324

- w złączeniach, 458

- zapis
  - brudny, 235
  - jednoczesny, 187
    - scalanie, 192
  - pojedynczego obiektu, 230
  - sekwencyjny, 84
  - zniekształcający, 245–248, 264
- zapobieganie
  - duplikatom, 499
  - utracie aktualizacji, 241, 243
  - zduplikowanym żądaniami, 500
- zapytania
  - analityczne, 238
  - deklaratywne, 56
  - o grafy, 64
  - w modelu MapReduce, 57
- zarządzanie
  - członkostwem i koordynacją, 358
  - współbieżnością
    - optymistyczne, 259
    - pesymistyczne, 259
  - złożonością, 35
- zasada punktów końcowych, 498, 500, 511
- zasoby współdzielone, 152
- zatwierdzanie
  - atomowe, 460
    - w jednym węźle, 344
    - w systemie rozproszonym, 344
  - dwuetapowe, 344
  - trzyetapowe, 349
- zdarzenia, 443
  - niemodyfikowalne, 444
- zegary, 283, 545
  - czasu rzeczywistego, 284
  - monotoniczne, 284
  - synchronizowane, 287, 290
  - wektorowe, 193
- złączanie
  - po stronie mappera, 396
  - przez scalanie, 398
  - strumieni, 455
  - z partycjami i haszowaniem, 397
  - z podziałem i haszowaniem, 416
  - z rozsyłaniem i haszowaniem, 396, 416
  - z sortowaniem przez scalanie, 393, 416
- złączenia, 391
  - strumień-strumień, 456, 463
  - strumień-tabela, 456, 463
  - tabela-tabela, 457, 463
- złożoność, 35
- znaczniki
  - czasu, 287
  - czasu Lamporta, 337
  - pól, 127

## Ż

- żądania
  - dotyczące wielu partycji, 504
  - trasowanie, 214

## O autorze

**Martin Kleppmann** jest badaczem systemów rozproszonych na Uniwersytecie Cambridge w Wielkiej Brytanii. Wcześniej był inżynierem oprogramowania i przedsiębiorcą w firmach internetowych takich jak LinkedIn i Rapportive, gdzie pracował nad działającą w dużej skali infrastrukturą do obsługi danych. Nauczył się przy tym kilku rzeczy na własnych błędach i ma nadzieję, że ta książka ochroni Cię przed popełnieniem podobnych pomyłek.

Martin jest blogerem, często występuje na konferencjach i uczestniczy w rozwijaniu oprogramowania o otwartym dostępie do kodu źródłowego. Wierzy, że ważne idee techniczne powinny być przystępne dla każdego i że lepsze ich zrozumienie pomoże w budowaniu lepszego oprogramowania.

## Kolofon

Zwierzę na okładce książki *Projektowanie aplikacji intensywnie przetwarzających dane* to dzik indyjski (łac. *Sus scrofa cristatus*) — podgatunek dzika zamieszkujący Indie, Myanmar, Nepal, Sri Lankę i Tajlandię. Różni się od dzika europejskiego tym, że ma dłuższą sierść na grzbiecie, jego sierść nie jest wełnista oraz ma większą i bardziej prostą czaszkę.

Dzik indyjski jest szaro lub czarno umaszczony i charakteryzuje się sztywną szczeciną wzdłuż kręgosłupa. Samce mają wystające kły (zwane szablami) służące do walki z rywalami i ochrony przed drapieżnikami. Samce są większe od samic. Średnia wysokość w kłębie wynosi dla gatunku 83 – 88 cm, a średnia waga to 90 – 140 kg. Naturalnymi wrogami dzików są niedźwiedzie oraz tygrysy i inne duże koty.

Dziki to zwierzęta nocne i wszystkożerne. Jedzą różny pokarm, w tym korzenie, owady, padlinę, orzechy, jagody i małe zwierzęta. Znane są też z tego, że ryją w śmieciach i polach uprawnych, powodując poważne zniszczenia i budząc wrogość rolników. Dziki potrzebują 4000 – 4500 kalorii dziennie. Mają dobrze rozwinięty zmysł węchu (co pomaga im znajdować rośliny i zwierzęta znajdujące się pod ziemią), jednak ich wzrok jest słaby.


Dziki od dawna zajmują ważne miejsce w ludzkiej kulturze. Według tradycji hinduskiej dzik jest awatarem boga Wisznu. Na starożytnych greckich nagrobkach był symbolem walecznego pokonanego (w przeciwieństwie do zwycięskiego lwa). Z powodu agresji dzik był przedstawiany na zbroi i elementach uzbrojenia wojowników skandynawskich, germańskich i anglosaskich. W chińskim zodiaku to zwierzę symbolizuje determinację i zapalczywość.

Wiele gatunków zwierząt prezentowanych na okładkach książek wydawnictwa O'Reilly jest zagrożonych. Wszystkie te gatunki są ważne dla świata. Aby dowiedzieć się więcej na temat tego, jak możesz pomóc, odwiedź stronę <http://animals.oreilly.com/>.

Grafika na okładce pochodzi z książki *Zoology* George'a Shawa.

# PROGRAM PARTNERSKI

— GRUPY HELION —

- 
1. ZAREJESTRUJ SIĘ
  2. PREZENTUJ KSIĄŻKI
  3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion** 

## KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE  
ZGODNIE Z METODĄ

# BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie  
z dostępem do nowoczesnych narzędzi - videokursów,  
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

**WWW.HELIONSZKOLENIA.PL**