

Teradata Vantage™ - Database Design

Release 17.10




July 2021

Copyright and Trademarks

Copyright © 2000 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to Database Design	10
Changes and Additions	10
Chapter 2: Database Design for Vantage	11
Advanced SQL Engine	11
Data Placement to Support Parallel Processing	12
Synchronization of Parallel Operations	15
Design Considerations	17
Databases and Data Modeling	27
Database Design Life Cycle	27
Designing for OLTP and Designing for Data Warehousing Support	27
ANSI/X3/SPARC Three Schema Architecture	28
Requirements Analysis	31
Logical Database Design	31
Activity Transaction Modeling	31
Physical Database Design	32
Chapter 3: Requirements Analysis	33
Developing an Enterprise Data Model	33
Chapter 4: Semantic Data Modeling	36
The Entity-Relationship Model	36
Entities, Relationships, and Attributes	36
Translating Entities and Relationships Into Tables	38
Relationship Theory	40
One-to-One Relationships	41
One-to-Many Relationships	42
Many-to-Many Relationships	42
Moving From an Entity-Relationship Analysis to Normalization	43
Chapter 5: The Normalization Process	44
Properties of Relations and Their Logical Manipulation	44
Functional, Transitive, and Multivalued Compatibilities	48
The Normal Forms	50
Third and Boyce-Codd Normal Forms	52
Decomposing Relations	56
Identifying Candidate Primary Keys	57
Foreign Keys	60
The Referential Integrity Rule	61

Domains and Referential Integrity	66
Normalization and Database Design Problems	68
General Procedure for Achieving a Normalized Set of Relations	73
Advantages of Normalization for Physical Database Implementation	74
Denormalized Physical Schemas and Ambiguity	78
Chapter 6: The Activity Transaction Modeling Process	86
Goals of the ATM Process	86
Terminology Used in the ATM Process	87
Example: The Location Entity Before It Is Fully Attributed	89
Example: A Randomly Selected Row From the Location Entity	89
Example: CustNum Column From the Location Entity	89
Example: The Primary Key Column for the Location Entity	90
Example: The Three Foreign Keys For the Location Entity	90
Domains	90
Column Names and Constraints	94
Guidelines for Naming Columns	96
Key Values and Relationships Among Tables	98
Domains Form	99
Constraints Form	103
System Form	105
Application Form	106
Report/Query Analysis Form	107
Table Form	112
Filling Out the Table Form	114
Table Form Example	114
Table Form: Basic Information	115
Table Form: Column-Level Information	116
Table Form: Miscellaneous Column-Level Information	116
Table Form: Access Information	117
Table Form: Data Demographics for Single-Column Database Objects	117
Maximum and Typical Column Value Frequencies	119
Table Form: Data Demographics for Multicolumn Database Objects	123
Row Size Calculation Form	127
Chapter 7: Denormalizing the Physical Schema	131
Denormalization Issues	131
Commonly Performed Denormalizations	133
Alternatives to Denormalization	133
Denormalizing with Repeating Groups	134
Denormalizing Through Prejoins	135
Denormalizing through Join Indexes	136
Derived Data Attributes	137
Denormalizing Through Global Temporary and Volatile Tables	138
Denormalizing Through Views	140

Dimensional Modeling, Star, and Snowflake Schemas	143
Chapter 8: Indexes and Maps	146
Primary Indexes and Primary AMP Indexes	146
Secondary Indexes	147
Join Indexes	148
Hash Indexes	149
Index Considerations	149
Index Type Comparisons	150
Evaluating Indexes	152
Maps	153
Chapter 9: Primary Index, Primary AMP Index, and NoPI Objects	157
Primary Indexes and Primary AMP Indexes	157
Primary Index Defaults	158
Unique and Nonunique Primary Indexes	159
Partitioned and Nonpartitioned Primary Indexes	161
Choosing an Indexing Method for a Column-Partitioned Table or Join Index	171
NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes	172
Column Partitioning	177
Column Partitioning Performance	179
Storage and Other Overhead Considerations for Partitioning	198
Advantages and Disadvantages of Partitioned Primary Indexes	199
Usage Recommendations For Row Partitioning	200
Single-Level Partitioning	217
Single-Level Partitioning Case Studies	226
Multilevel Partitioning	227
Three-Level Row Partitioning Example	240
Summary of Primary Index Selection Criteria	245
Principal Criteria for Selecting a Primary Index	245
Selecting a Primary Index for a Queue Table	251
Column Distribution Demographics and Primary Index Selection	252
Scenario 1	255
Scenario 2	259
Scenario 3	260
Scenario 4	262
Performance Considerations for Primary Indexes	264
Duplicate Row Checks for SET Tables with NUPIs	266
Minimizing Duplicate NUPI Row Checks	266
Chapter 10: Secondary Indexes	267
Space Considerations	267
Unique Secondary Indexes	267
Nonunique Secondary Indexes	269
NUSI Bit Mapping	272

NUSIs and Query Covering	273
Value-Ordered NUSIs and Range Conditions	274
Selecting a Secondary Index	277
Secondary Index Access Summarized by Example	280
Chapter 11: Join and Hash Indexes	285
Join Indexes	285
Using Join Indexes	294
Join Index Design Tips	303
Join Index Benefits and Costs	305
Maintenance Cost as Function of Hits for Each Data Block	312
Maintenance Cost as a Function of Row Size	313
Join Index Maintenance Cost as a Function of Insert Method	315
Cost/Benefit Analysis of Join Indexes	318
Cost/Benefit Analysis for Join Indexes	318
Join Index Types	322
Simple Join Indexes	323
Defining a Simple Join Index on a Binary Join Result	325
Defining and Using a Simple Join Index With an <i>n</i> -way Join Result	327
Single-Table Join Indexes	328
Single-Table Join Index	333
Aggregate Join Indexes	335
Sparse Join Indexes	339
Using Outer Joins in Join Index Definitions	341
Using Outer Joins to Define Join Indexes	342
Creating Join Indexes Using Outer Joins	344
Join Indexes and Tactical Queries	347
Join Index Definition Restrictions	356
Improving Join Index Performance	378
Join Index Storage	382
Value-Ordered Storage of Join Index Rows	385
Hash Indexes	386
Collecting Statistics on Hash Index Columns	392
Hash Index Definition Restrictions	394
Hash and Join Index Interactions With Other Teradata Systems and Features	395
Tradeoffs for Join or Hash Indexes	397
Chapter 12: Designing for Database Integrity	399
Sources of Data Quality Problems	399
Logical Integrity Constraints	403
How Relational Databases Are Built From Logical Propositions	407
Inclusion Compatibilities	408
Semantic Integrity Constraint Types	409
Semantic Constraint Specifications	412
Semantic Constraint Enforcement	429

Updatable Cursors and Semantic Database Integrity	430
Semantic Integrity Constraints for Updatable Views	431
Summary of Fundamental Database Principles	435
Physical Database Integrity	436
Disk I/O Integrity Checking	437
About Reading or Repairing Data from Fallback	438
Chapter 13: Designing for Missing Information	439
Semantics of SQL Nulls	439
Inconsistencies in How SQL Treats Nulls	439
Bivalent and Higher-Valued Logics	441
Alternatives To Nulls for Representing Missing Information	443
Systematic Use of Default Values	444
Redesigning the Database to Eliminate the Need for Nulls	444
Manipulating Nulls With SQL	446
Logical and Arithmetic Operations on Nulls	446
NULL Literals	450
Hashing on Nulls	451
Null Sorts as the Lowest Value in a Collation	452
Searching for Nulls Using a SELECT Request	452
Searching for Nulls and Nonnulls In the Same Search Condition	452
Excluding Nulls From Query Results	452
Nulls and the Outer Join	453
Chapter 14: Using Data Compression	456
Compression Types Supported by Vantage	456
Identifying Uncompressed, Single-Value Compressed, and MultiValue Compressed Tables	458
Multivalue Compression	460
Tradeoffs Between Multivalue Compression and Storage Requirements for Compressed Values	463
Algorithmic Compression	464
Row Compression	467
Row Header Compression	468
Autocompression	468
Using Hash Index and Join Index Row Compression	469
Block-Level Compression	471
Hardware-Based Block-Level Compression	471
Software-Based Block-Level Compression	472
Compressing Data Loaded into Empty Subtables Set to AUTOTEMP	476
Interaction between Block-Level Compression and TVS Temperature Query Band Values	477
Controlling BLC at the System Level Using the DBS Control Utility	477
Block-Level Compression Usage Notes	483
Interaction between Block-Level Compression Settings	484
CPU Considerations on Different Teradata Platforms	484
Finding Tables and Databases with Compressed Data Blocks	484
Obtaining Information about Tables with Compressed Data Blocks	485

About System and AMP Outages During Compression	487
CPU Costing for Software-Based BLC	488
Estimating BLC Space Savings and CPU Usage	488
Choosing a Software-based Compression Scheme	488
Combining Compression Methods	489
Related Information	489
Chapter 15: Database-Level Capacity Planning Considerations	490
Capacity Planning	490
Storing Data Efficiently	493
Base Table Row Format	496
Hash and Join Index Row Structures	522
Presence Bits	528
Table Headers	533
Sizing Structured UDT Columns	539
System-Derived and System-Generated Columns	544
Data Type Considerations	547
Numeric Data Types	548
Integer Data Types	550
Non-INTEGER Numeric Data Types	551
Byte Data Types	554
DateTime Data Types	554
Interval Data Types	556
Period Data Types	561
Character Data Types	564
XML/XMLTYPE Data Type	566
JSON Data Type (Text-Based Format)	566
JSON Data Type (Binary Format)	567
JSON Data Type (Universal Binary Format)	568
DATASET Data Type	568
User-Defined Data Types	569
Array Data Types	569
Row Size Calculation	570
Sizing Databases, Users, and Profiles	576
Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables	583
Sizing Base Tables, Hash Indexes, and Join Indexes	584
Sizing a LOB or XML Subtable	586
Sizing a Unique Secondary Index Subtable	587
Sizing a Nonunique Secondary Index Subtable	588
Sizing User-Defined Routines	590
Sizing a Reference Index Subtable	590
Sizing Spool Space	592
Sizing a Query Capture Database	594
Sizing Table Space Empirically	596

Chapter 16: System-Level Capacity Planning Considerations	599
Database Size Considerations	599
Teradata Secure Zones inside a Database	599
System Disk Contents	600
Data Disk Contents	601
Data Disk Space	602
Permanent Space Allocations	602
Estimating Database Size Requirements	604
About Global Space Accounting	606
Determining Available User Table Data Space	609
Designing for Backups	615
Chapter 17: Design Issues for Tactical Queries	618
Tactical Queries Defined	618
Scalability Considerations for Tactical Queries	618
Localizing the Work	621
Database Design Techniques to Support Localized Work	628
Single-AMP Queries and Partitioned Tables	630
Recommendations for Tactical Queries and Row-Partitioned Tables	631
Sparse Join Indexes and Tactical Queries	632
All-AMP Queries	633
All-AMP Tactical Queries and Partitioned Tables	634
Application Opportunities for Tactical Queries	635
Other Tools Useful for Monitoring and Managing Tactical Queries	638
Monitoring Active Work	639
Appendix A: Notation Conventions	641
Appendix B: Teradata System Limits	644
Appendix C: Designing With Task-Oriented Profiles	661
Appendix D: Summary Physical Design Scenario	665
Appendix E: Sample Worksheet Forms	666
Appendix F: Designing Tables for Optimal Performance	677
Appendix G: Compression Methods	681
Appendix H: References	688
Appendix I: Additional Information	689

Introduction to Database Design

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

This document describes database design for Vantage, including logical design and physical design, capacity considerations, and designing for tactical queries.

Changes and Additions

Date	Description
July 2021	Fallback subtables are no longer compressed by default.

Database Design for Vantage

A properly designed and configured database can provide ready access to the enterprise-wide data to answer the ad hoc, tactical, decision support, data mining, and analytic requests that a modern business enterprise generates in its quest to remain ahead of its competition.

This section discusses aspects of the Vantage architecture that need to be understood for database design and provides general design considerations. This section also introduces the concepts of database modeling, both logical and physical, as a preview for the remaining sections in this document.

Advanced SQL Engine

Goals

- A large capacity, parallel processing database system with thousands of MIPS capable of storing terabytes to petabytes of total user data and billions of rows in a single table.
- Fault tolerance, with no single point of failure, to ensure data integrity.
- Redundant network connectivity to ensure system throughput.
- Manageable, scalable growth.
- A fully-relational database management system using a standard, non-proprietary access language.
- An extensive and extensible set of data types including support for structured, semi-structured, and unstructured data.
- Faster response times than any competing relational database management systems.
- A centralized shared information architecture in which a single version of the truth is presented to users.

There is a difference between a single version (or source) of the truth and a single view of the truth. It is quite possible, and often very necessary, to have multiple views of the truth, but these multiple views should all be based on a single version of the truth if they are to be relied upon for decision making.

Architected for Parallel Processing

Because Vantage was designed to perform parallel processing from the outset, the Teradata system architecture does not suffer from the allocation of shared resources that other system that have been adapted for parallelism experience. This is because the system is designed to maximize throughput, while multiple dimensions of parallel processing are available for each individual system user. Repeating for emphasis: the Vantage architecture is parallel from the ground up and has always been so. Its file system, message subsystem, lock manager, and query optimizer all fit together snugly, all working in parallel.

The Vantage parallel technology is optimized to perform tasks in a normalized environment that other relational DBMSs cannot match with a denormalized schema. Teradata is also optimized to perform tasks in a denormalized environment. Among the special performance advantages built into the Teradata system are the following: star join and other join optimizations, full-table scan optimization, specially designed index

types, a full complement of SQL aggregate and ordered analytical functions, and the most sophisticated parallel-aware SQL query optimizer available.

Among the fundamental aspects of Teradata parallelism are:

- [Data Placement to Support Parallel Processing](#)
- [Synchronization of Parallel Operations](#)

Data Placement to Support Parallel Processing

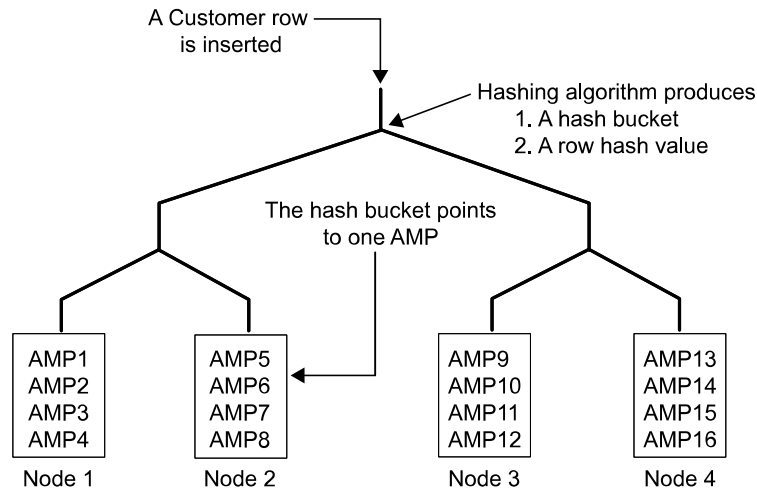
Balanced Workloads

The key to parallel processing is balancing the workload of the database management system. SQL Engine balances its workload by distributing table rows evenly across the AMPs and by giving them the sole responsibility for the data they own.

Row Distribution

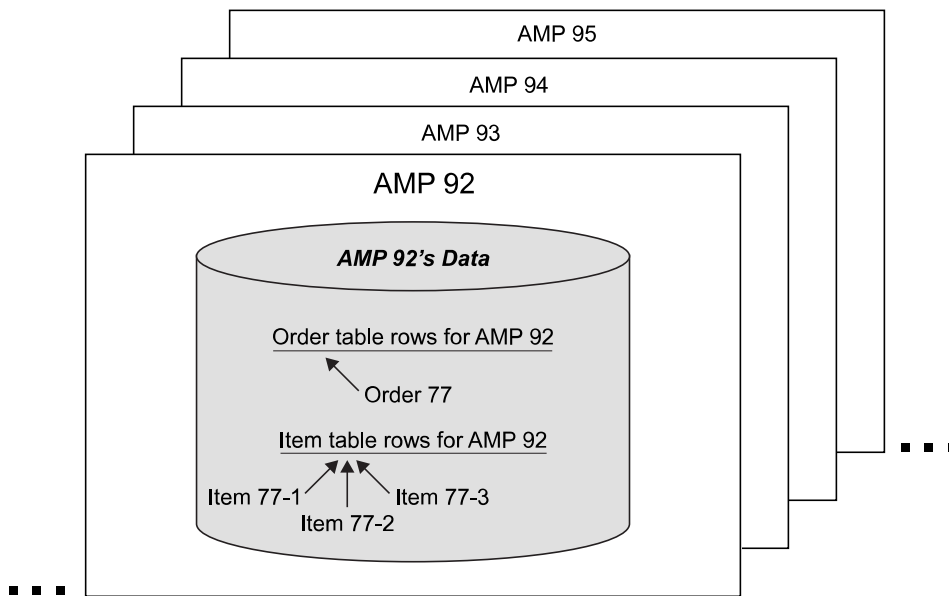
Database rows are hashed across a table's AMPs using the row hash value of their primary index or primary AMP index as the hash key. This does not apply for the rows of NoPI tables.

Vantage also uses the row hash of a primary index or primary AMP index to retrieve a row. The following graphic shows how a row is hashed and assigned to an AMP:



By carefully choosing the primary index or primary AMP index for each table, you can ensure that rows that are frequently joined hash to the same AMP, eliminating the need to redistribute the rows across the BYNET in order to join them.

The following graphic shows how you can set up rows from commonly joined tables on the same AMP to ensure that a join operation avoids being redistributed across the BYNET:

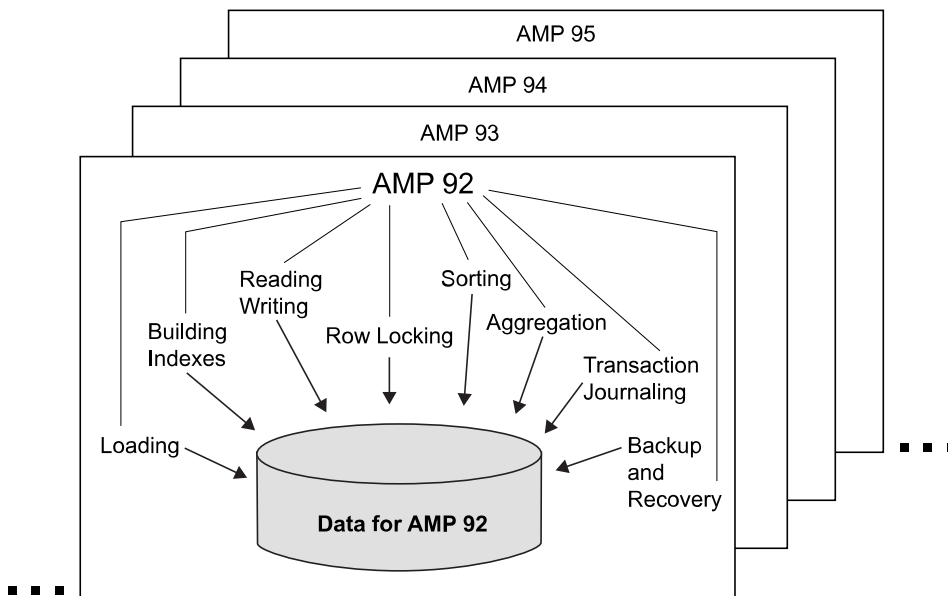


Shared-Nothing Architecture

The term *shared-nothing architecture* is used to describe a multiprocessor database management system in which neither memory nor disk storage is shared among the processors. The PE and AMP vprocs in the SQL Engine architecture share neither memory nor disk across CPU units; therefore, Vantage uses a shared-nothing database architecture. It is this architecture that affords Teradata systems their scalability.

AMP Ownership of Data

Because of its shared-nothing architecture, each AMP in a Teradata system exclusively controls its own virtual disk space. As a result, each row is owned by exactly one AMP. That AMP is the only one in the system that can create, read, update, or lock its data. The AMP-local control of logging and locking not only enhances system parallelism, but also reduces BYNET traffic significantly. The following graphic shows how local autonomy provides each AMP (AMP 92 in this particular example) with total accountability for its own data.

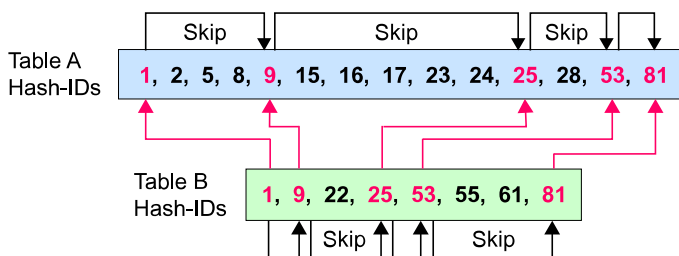


Other Applications of Hashing for Parallel Processing

SQL Engine employs row hashing for tasks beyond simple row distribution and retrieval.

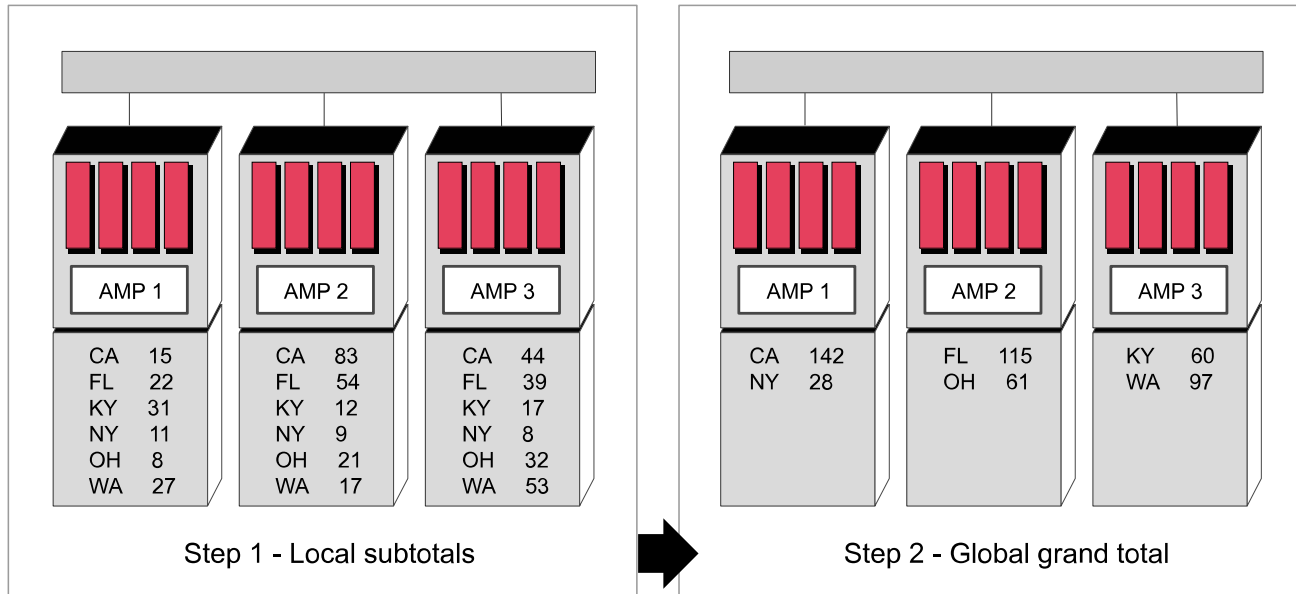
Several types of join processing hash on join column values to determine which AMP is to handle the join of the particular rows. This enables Vantage to balance the join load across AMPs, with each doing an even subset of the total work.

Similarly, the row hash match scan method sorts rows to be joined on their row hash values, then scans the joined tables in parallel to permit the scan of one of the tables to skip ahead in its hash scan to the row hash where the second table is already positioned, as indicated by the following graphic:



In this example, Table A reads and joins row 1 to row 1 of Table B. Table A then obtains the row hash value of the next row in Table B, row 9, and joins its row 9 to row 9 of Table B, which has a matching row hash value. Unmatched rows are skipped without being read by hashing to the next highest candidate value for which a join might be possible. This processing of candidate joined rows can shorten the time required to perform this join significantly. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information about this join method.

Aggregate processing also takes advantage of hashing to build its totals. For example, the following graphic indicates how each AMP aggregates its rows locally as the parallel first step in the global parallel aggregation process:



Following that, the fields in the GROUP BY key are hashed, and the resulting hash bucket for each distinct value points to the AMP responsible for building the global aggregate for that piece of the aggregate.

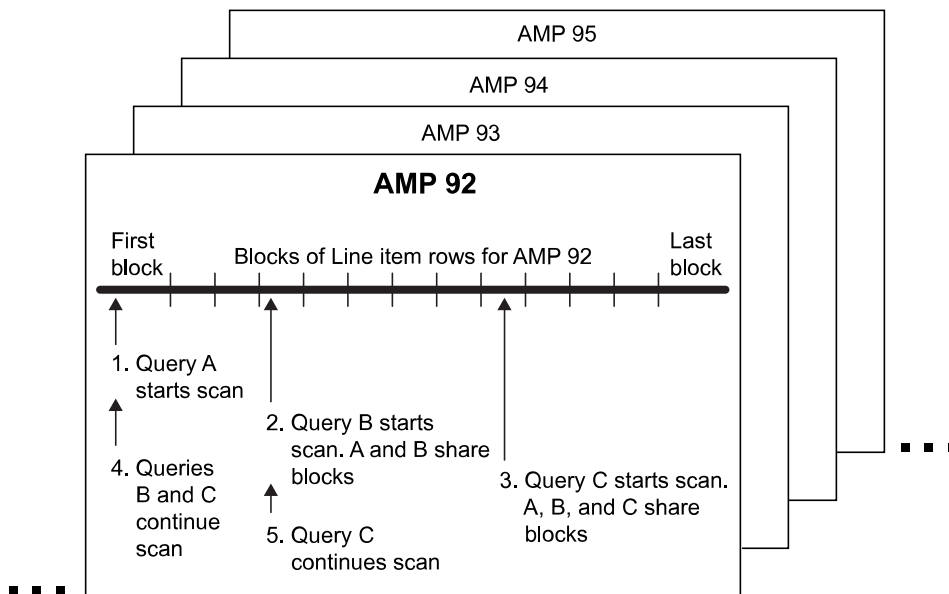
The illustration shows only 1 AMP per node for simplicity of presentation. A real Teradata system typically has multiple AMPs per node, and each would be involved in performing its own role in producing the global aggregate totals.

Synchronization of Parallel Operations

Synchronized Table Scans

Teradata system architecture is not only inherently parallel, but it is also highly optimized to synchronize its parallel operations to effect further performance optimizations.

The database can perform synchronized scanning of cached large tables. Synchronized scanning permits new full-table scans to begin at the current point of an ongoing scan of the same table, thus avoiding any I/O for the subsequent scans. Because synchronized scans might not progress at the same rate, any task can initiate the next read operation.



Spool Reuse

Vantage reuses the intermediate answer sets referred to as spools within a request if they are needed at a later point to process the same query. Two common examples of spool reuse are table self-joins and correlated subqueries.

Synchronized BYNET Operations

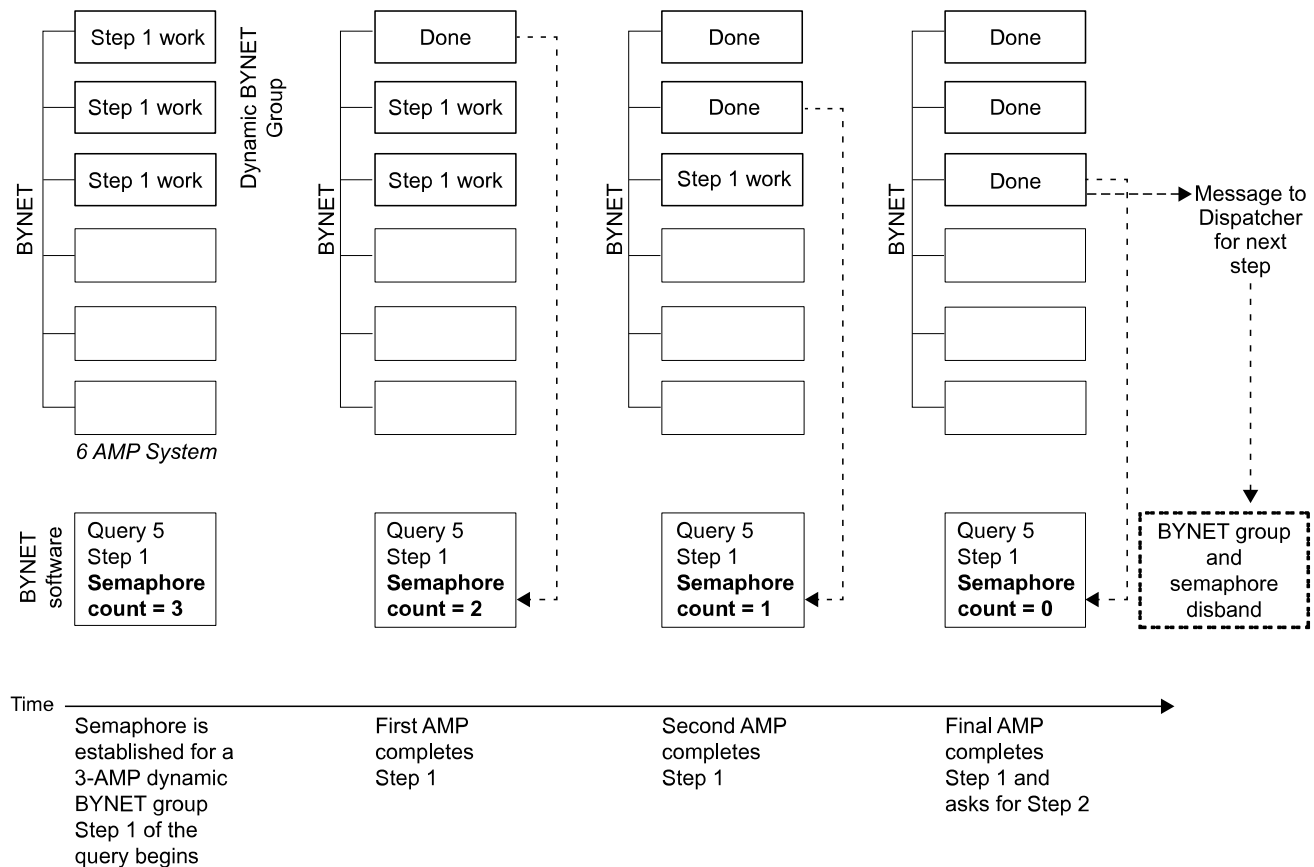
Several Vantage features act to minimize message flow within the SQL Engine. Primary among them are dynamic BYNET groups and global semaphores.

A dynamic BYNET group is an ad hoc grouping of the AMPs working on a single step. Any number of dynamic BYNET groups can coexist simultaneously. Dynamic BYNET groups are established when the first step of an optimized request is about to be dispatched to the AMPs.

Before the first step is dispatched, a message is sent via the BYNET to only those AMPs that will be involved in processing the step. As a result, a dynamic BYNET group might be composed of a single AMP, a group of AMPs, or all AMPs in the system. When an AMP acknowledges reception of the message, it is enrolled in the dynamic BYNET group, sparing the database software from having to initiate a separate communication.

Success and failure global semaphores are associated with dynamic BYNET groups. These objects exist in the BYNET software and signal the completion (or failure) of a step by the first and last AMPs in the dynamic BYNET group. Each success semaphore maintains a count of the number of AMPs in the group still processing the step. As each AMP reports its success, the semaphore count is reduced by 1, so when the last AMP in the group reports, the value of the semaphore is 0. Each AMP reads the value of the success semaphore when it finishes processing the step, and when its value reaches 0, that AMP knows it is the last in the group to complete the task. At that point, the AMP sends a message back to the Dispatcher to notify it that all AMPs in the dynamic BYNET group have finished processing the step, so it can then send the next

step to the group. This is the only message sent to the Dispatcher for each step irrespective of the number of AMPs in the dynamic BYNET group.



Failure semaphores are the mechanism the SQL Engine uses to support global abort processing. A failure semaphore signals to the other AMPs within a dynamic BYNET group that one of its members has experienced a serious error condition so they can abort the step and free system resources immediately for other uses.

With success and failure semaphores, no external coordinator is required. Global synchronization is built into the Teradata system architecture, and because there is no single point of control, performance can be scaled up to easily handle an increasing volume of system users.

Design Considerations

Certain aspects of commonly recommended database design strategies should be evaluated for their veracity before you begin designing the database that supports your business. Some typical uses cases and their usage consideration are discussed in the following sections.

- [Usage Considerations: OLTP and Data Warehousing](#)
- [Usage Considerations: Summary Data and Detail Data](#)
- [Usage Considerations: Simple and Complex Queries](#)

- [Usage Considerations: Ad Hoc Queries](#)

Usage Considerations: OLTP and Data Warehousing

The two common commercial uses for relational database management systems are online transaction processing (OLTP) and data warehousing (DW). The access patterns of these two approaches are very different and they make very different demands on the underlying SQL Engine.

The next few topics examine the different access patterns of OLTP and DW processing. Once the two styles of processing have been compared, you should readily recognize just how dramatically different they are.

OLTP

Consider the following simple example: A customer, Mr. Brown, orders a calendar. The graphic diagrams the actions taken against the database.

Item

Item #	Quantity	Description
PK		
3509	10	Calendar
3360	935	8.5 x 14 paper
3474	0	Stapler
3345	875	8.5 x 11 paper
3421	0	#2 pencil

Customer

Customer #	Name	Phone
PK		
2	James	555-4444
3	Brown	444-3333
12	Adams	111-9999
9	Black	444-5555
13	Rice	888-9999

Order

Order #	Customer #	Order Date	Status
FK	FK		
7324	2	Mar 13	0
7325	3	Mar 13	0
7326	3	Apr 5	0

Order Item Shipped

Order #	Item #	Quantity	Ship Date
PK			
FK	FK		
7324	3360	100	Mar 14
7325	3509	30	Mar 14
7324	3474	30	Mar 14
7324	3421	0	Mar 14
7325	3474	10	Mar 14
7326	3509	1	Apr 5

Order Item Backordered

Order #	Item #	Quantity
PK		
FK	FK	
7324	3421	144
7325	3474	10

The transaction flow is outlined in the following process stages:

1. A new order, 7326, is opened on the Order table.
2. The remaining columns for the order are filled out, including the Customer number for Mr. Brown, 3, and the date the order was placed, April 5.
3. A new shipping order, with order number 7326 and item number 3509, is opened on the Order Item Shipped table.
4. The remaining columns for the shipper are filled out, including the quantity shipped, 1, and the date the order was shipped, April 5.
5. The transaction is complete.

Three key aspects of this transaction deserve your attention:

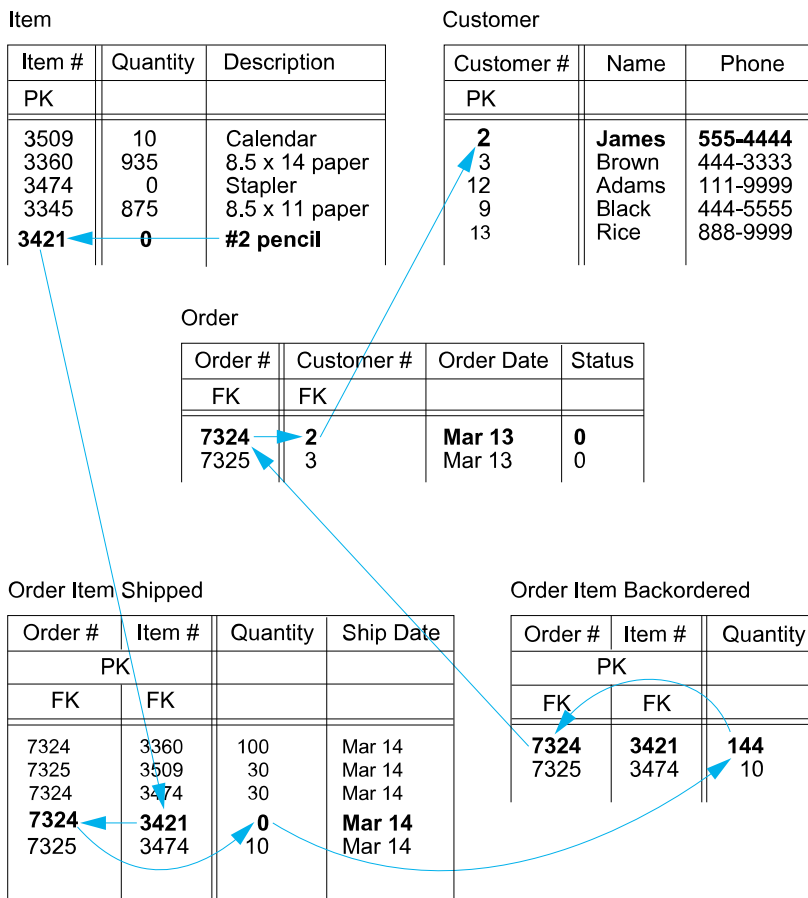
- Only a few of many possible tables were accessed.
- None of the accessed tables, some of which might have billions of rows, were scanned.

- Very little I/O processing was required to complete the transaction.

These three attributes are characteristic of OLTP environments. The requirements are simple and involve only a quick-in, quick-out approach to the database. Many relational database engines are designed specifically for maximum efficiency with these kinds of workloads.

Data Warehousing

Consider the following simple decision support example using the same database to emphasize the contrast in processing requirements: A business analyst needs to find the answer to a specific question about pencil customers so he can target them for a promotion. The natural language query the analyst performs is this: which customers placed the majority of their orders for pencils in the first quarter of this year? The graphic diagrams the actions taken against the database.



The processing flow is outlined in the following process stages:

1. The SQL Engine interrogates the Item table for the item number for pencils, which is 3421.
2. The engine next scans the Order Item Shipped table looking for all orders associated with item number 3421.
3. The particular row examined in this example shipped 0 pencils, indicating that the order was back ordered.

4. The engine accesses the Order Item Backordered table row having an order and item number matching that examined in stage 3 and finds that 144 pencils were back ordered.
5. The engine must now locate the customer who placed this order because the whole point of the query was to identify customers who had ordered pencils in the first quarter of the year.

To do this, the engine must access the Order table, where it finds that customer number 2 placed the order in question.

6. The engine accesses the Customer table to identify the name of the customer associated with customer number 2, which is James.

There are also three key aspects of this query that deserve your attention:

- The large number of tables that had to be accessed in order to answer it.
This complex access path through many tables is a hallmark of decision support analysis. And remember: this is an extremely simple example!
- The tables were not just touched lightly in response to the query, but required massive searches, and sometimes multiple scans are required.
- Once the data required to answer the query has been gathered, it must be processed further using aggregation, joins, sorts, conditional requirements, and so on.

All this processing is heavily resource-intensive. The following table presents a summary contrast of DW and OLTP processing.

OLTP Transaction	Attribute	DW Query
No	Multiple tables scanned	Yes
No	Large volumes of data examined	Yes
No	Processing-intensive	Yes
No	Response time is a function of database size	Yes

Usage Considerations: Summary Data and Detail Data

This topic examines the nature of the data you keep in your data warehouse and attempts to indicate why storing detail data is a better idea, particularly for ad hoc tactical and decision support queries and data mining explorations.

Observing the Effects of Summarization

Suppose we have an application that gathers check stand scanner data and stores it in relational tables. The raw detail data captured by the scanner includes a system-generated transaction number, codes for the individual items purchased as part of the transaction, and the number of each item purchased. The table that contains this detail data is named Scanner Data in the following graphic:

Scanner Data			Store Item <u>D</u> aily Sales				Store Item <u>W</u> eekly Sales			
Checkout Transaction No.	Item No.	Quantity Sold	Store No.	Item No.	Date	Quantity Sold	Store No.	Item No.	Week Ending	Quantity Sold
PK			PK				PK			
FK	FK	NN	FK			NN, DD	FK			NN, DD
1234001	1563	12
1234001	807	1	1	2	Jun 01	110	1	2	Jun 07	1363
1234001	2	1	1	2	Jun 02	126
1234001	149	4	1	2	Jun 03	127	2	2	Jun 07	456
...	1	2	Jun 04	144
...	1	2	Jun 05	102
1234005	402	3	1	2	Jun 06	344
1234005	2	2	1	2	Jun 07	410
...
...	2	2	Jun 01	50
1234027	2	3	2	2	Jun 02	47
1234027	807	3	2	2	Jun 03	32
...	2	2	Jun 04	20
...	2	2	Jun 05	37
1234046	177	6	2	2	Jun 06	144
1234046	807	1	2	2	Jun 07	126
1234046	2	3
...
1234639	2	1
1234639	177	1
...
...

The middle table, *store_item_daily_sales*, illustrates a first level summary of the data in *scanner_data*. Notice that where we knew which items sold at which store at which time of day in *scanner_data*, now we only know the quantity of an item sold for an entire business day. The clarity of the detail data has been replaced by a more fuzzy snapshot. Potential insights have been lost.

The right most table, *store_item_weekly_sales*, illustrates a further abstraction of the detail data. Now all we have is the quantity of each item sold by each store for an entire week. Still fewer insights can be garnered from the data.

Of course, the data could be further abstracted. Summarization can occur at many levels. The important lesson to be learned from this study is that summaries hide valuable information. Worse still, it is not possible to reverse summarize the data in order to regain the original detail. The sharper granularity is lost forever.

Consider this simple, and highly logical, query that an analyst might ask of the sales data: How effective was the mid-week promotion we ran on sales for an item on Tuesday and Wednesday? If the only data available for analysis is a unit volume by week entity, then it is not possible to answer the question. The answer to the question is to be found in the detail, and the analyst has no way to determine the effectiveness of the promotion.

Other basic questions that cannot be answered by summary data include the following:

- What is the daily sales pattern for item 2 at any given store?

- When a customer purchases item 2, what other items are most frequently purchased with it?
- What is the profile of a customer who purchases item 2?

Information Value of Summary Data

The information value of summary data is extremely limited. As we have seen, summary data cannot answer questions about daily sales patterns by store, nor can it reveal what additional purchases were made in the same market basket, nor can it tell you anything at all about the individual customer who made the purchase.

What summary data can provide is summary answers and nothing more. This puts you in the position of always being reactive rather than proactive. Two classic retail dilemmas posed by this summary-only situation indicate that both extremes of a given problem can be caused by only having access to summary data:

- An out-of-stock situation has only been discovered after it is too late to remedy the problem.
- There is too much stock on hand, forcing an unplanned price reduction promotion to eliminate the unwanted inventories.

Proactive Use of Detail Data

Suppose the retailer in this case example used detail data to analyze which products tend to cluster in the same market basket. Once the product clusters have been determined, it is possible to rearrange the layout of shelf displays to encourage yet more of this buying behavior.

As another example, a retailer could use detail data to determine what types of customer tend to buy a particular product or product family. With this information in hand, the retailer could then target a specialized promotion to cross-sell those customers on other products.

Usage Considerations: Simple and Complex Queries

Users query their databases not because they need an answer as an end result; rather, they want guidance for performing a business action that will have optimal decisive effects. The actions taken cause your bottom line to change, not the answers that informed those actions.

A query can be simple or it can be complex. The answers provided by simple queries tend to be more expected than not, while the answers provided by complex queries tend to produce far less certain answers that are that much richer for resolving the questions they pose.

Consider the following case example from the financial industry. A bank offers its customers a financial instrument that a competitor has decided to offer “without charge.” This challenge must be responded to quickly or the bank will lose many of its customers who are currently paying what they see as an unnecessary fee for the financial instrument under discussion.

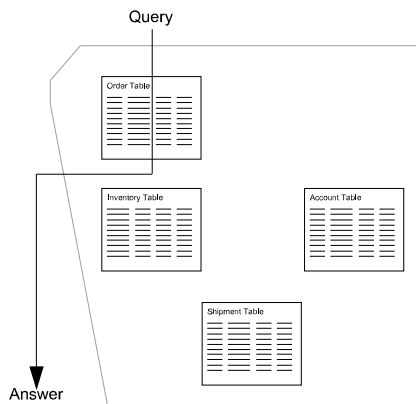
Suppose the bank responds to this situation by asking a simple question such as “which current customers use this product”? The most likely response to this information, which is merely a list of current customers, would be to eliminate the fee for the product. While this action is likely to forestall the erosion of the customer base, it also has the negative effect of reducing profits for the bank.

Suppose, instead, that the bank asks a more sophisticated question such as “which current customers using this product would remain profitable clients even if the fee were eliminated”? The bank now knows not only which consumers of its financial instrument are profitable for reasons other than their consumption of that product, but also knows which consumers do not otherwise contribute to the bottom line. The latter customer set can be released to the competition, which, by the way, almost certainly does not know that the new customers it is luring away from our bank are not profitable.

The impact of this more complex query is profit maximization, and its example illustrates very clearly the value of complex over simple queries.

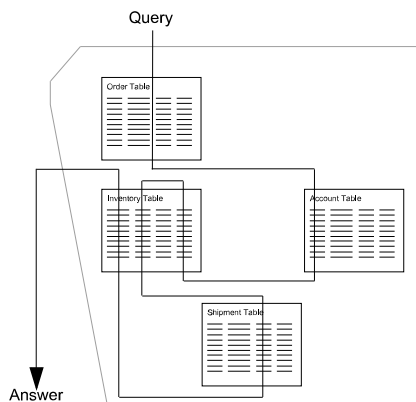
Relationship Between Query Complexity and the Value of Its Answer

A simple query typically accesses only a few tables, as illustrated by the following graphic.



The simplicity of such a query maps directly into the simplicity of the answer it returns. In other words, simple queries tend to deliver low-value answers which, in turn, enable low-value actions.

More complex queries, on the other hand, investigate the multivariate relationships among many tables in search of the high-value answers that come from mining the many interrelationships among the tables accessed. The following graphic illustrates the concept of a moderately complex query. The example winds its way through four individual tables, accessing one of them several times. A more realistic complex query could easily access hundreds of tables, including as many as 128 of them in a single join!



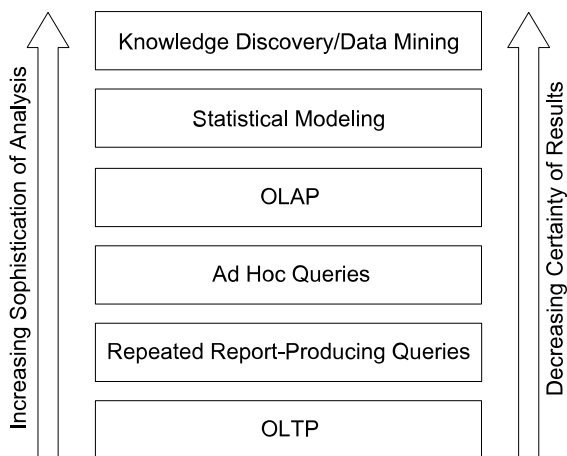
Query complexity exerts a processing and I/O burden that many commercial relational database management systems are not capable of handling, which is the principal reason that most data warehouse vendors advocate the use of summary data.

The relationship between query complexity and the extent of detail in the database is direct and profound. While summary data is often good enough to answer simple queries, it cannot deliver the sorts of answers that more complex queries seek. This is an extremely important concept to understand before designing your databases, because you need to provide the level of detail in the data that can deliver answers to the types of questions you will be asking.

Valuable Information and Time

The value of information is often inversely proportional to the length of time required to derive it.

As the following graphic illustrates, the more sophisticated the analysis, the less expected the answers obtained. More explicitly, this principle can be stated as the more complex the query, the more likely that heretofore unknown information hidden in the data is revealed.



Sophisticated explorations of the data universe might take longer to produce results, but when those results are finally produced, they are invaluable to the business. A quick response is a minimal requirement for simple queries, but such queries rarely provide a business-critical response, and designing a database to ensure nothing but quick responses is the quickest path to failure for your data warehouse project. The parallel architecture built into Vantage ensures that all queries are answered in an optimal time frame.

Your data warehouse can be a source of unimaginable information richness, but only if it is designed with the thought that any possible question, no matter how involved or abstract, should be possible to answer as readily as a simple query.

Usage Considerations: Ad Hoc Queries

In the data warehousing world, the phrase *ad hoc query* generally refers to a query composed at the keyboard for immediate performance (as opposed to a permanent query stored in a macro, stored procedure, or SQL application that is performed over and over again without alteration). Such a query

frequently undergoes careful, extended planning before being performed for the first time and then is often revised and refined interactively until the desired response is returned. The goal of this type of ad hoc query is not the reporting of data, but the discovery of information.

Ad Hoc Queries and Enterprise Tactical Support

Tactical queries are focused on operational decision making rather than enterprise discovery or bookkeeping activities. They typically have the following characteristics:

- Relatively simple syntax
- Direct access
- Highly tuned
- Have expected response latencies on the order of 20 seconds or less, and are often expected to return a response in less than one second.

See [Design Issues for Tactical Queries](#) for many of the special considerations you must take into account when designing to support tactical queries.

Ad Hoc Queries and Enterprise Discovery

[Usage Considerations: Simple and Complex Queries](#) established that discovery is the critical focus of data warehousing, not simple reporting. The trends fueling the data warehousing revolution are based on the need for information that can be acted upon proactively. Report-oriented systems rely on predefined questions that produce reports of what has already happened. This type of information can only produce evidence that a situation that has already occurred must be reacted to.

A data warehouse must have the capability of performing high performance, complex, ad hoc queries if it is to capitalize on the advantages of warehousing its business data. Furthermore, that data must be available for analysis in detail, as well as summary, formats. The value of data warehousing comes from being able to ask unplanned questions on detail data.

Business Situations That Drive the Need for Ad Hoc Queries

Two general situations typically drive ad hoc queries: exploratory analysis to discover business opportunities and detailed analysis of why some complicated event that had a negative impact on the enterprise occurred. You must be able to ask questions as they present themselves in either situation and the data warehouse you use for your analysis must be capable of accommodating that need.

Consider the search for new business opportunities. The following table lists some likely business opportunity searches by industry:

Industry	Typical Query Focus
Financial	Cross-selling opportunities
Retail	Market basket analysis
Communications	Calling patterns indicating high risk of losing a customer to the competition.

Now consider the “what went wrong” type of analysis. The following table lists some likely scenarios for this type of ad hoc query by industry:

Industry	Typical Query Focus
Transportation	<ul style="list-style-type: none"> • Over capacity • Under capacity
Communications	Sudden customer attrition

Businesses cannot know what questions they will need answered in the future, but they must be able to ask questions that permit them to influence the future positively. Neither issue can be dealt with by means of an analysis of the data warehouse if the system does not provide support for complex ad hoc queries.

Databases and Data Modeling

The purpose of this document is to provide guidelines to help you design a database that provides users with the following characteristics:

- A single, unified, unambiguous view of the data
- Optimal access for any possible valid query
- Freedom from navigation

These attributes can be achieved by careful collection and analysis of requirements and through careful planning of how to implement those requirements. The name used to describe the planning of the structure and relationships of a database is data modeling.

Data modeling comes in two modes: logical and physical.

These two modes can be done one after the other or interleaved in an agile fashion depending on your requirements and implementation style.

Database Design Life Cycle

Database design is a living process: it never ends. Not only do new features and optimizations come with each new software release, but new business needs and supporting information requirements continually present themselves and must be integrated with the production database.

In some circumstances, the integration process can be as involved as re-engineering entire components of the database. Fortunately, if you have adhered to an implementation that is fully normalized, the re-engineering process is a relatively harmless and predictable activity.

Designing for OLTP and Designing for Data Warehousing Support

Differences Between OLTP and Data Warehouse Processing

The following table examines some of the principal differences in OLTP and Data Warehouse processing that are critical to database design.

Attribute	Processing Type	
	OLTP Transaction	Data Warehouse Query
Multiple tables scanned?	No	Yes
Large data volumes?	No	Yes
Processing intensive?	No	Yes
Processing time a function of database size?	No	Yes

As you can see from this side-by-side comparison, OLTP transactions and Data Warehouse queries are dramatically different entities and, by inference, might be expected to require physically different database support.

Data Warehousing Support

Relational databases that support data warehousing are optimized to support decision support applications in all the forms listed below and more.

- Ad hoc SQL or natural language-generated SQL queries
- Repeatedly stated queries in the form of macros, embedded SQL applications, or stored procedures
- Data mining and OLAP explorations

The data in the warehouse is stored in the form of multiple normalized tables that model your enterprise. The decision support applications that explore these data often perform full file scans of multiple large tables, making them processing intensive. Because processing time is directly related to data volume, the responsiveness of the system is heavily affected by the size of the underlying database.

OLTP

OLTP transaction processing operates differently. OLTP systems do not scan multiple tables simultaneously, nor do they access large volumes of data. This makes their consumption of processing and I/O resources minuscule in comparison with decision support query processing. Size is not a factor in system responsiveness.

ANSI/X3/SPARC Three Schema Architecture

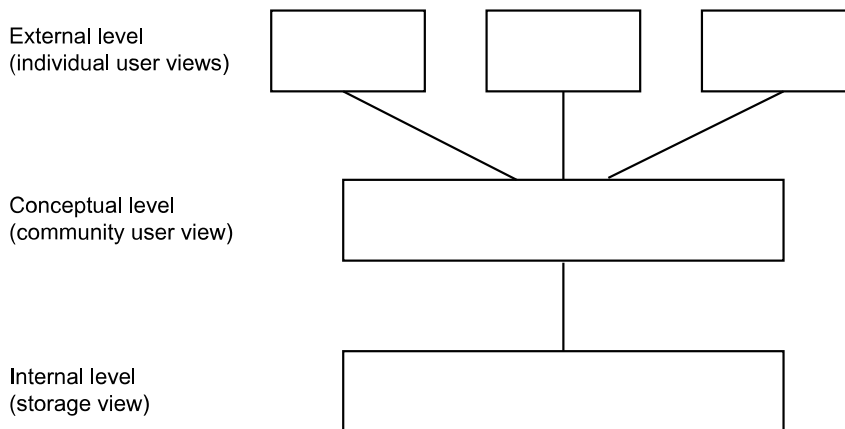
High-Level Architecture

The following graphic presents a high-level (and highly simplified) view of the ANSI/SPARC architecture.

ANSI/SPARC specifically names three levels of the architecture.

- External
- Conceptual
- Internal

The External level is composed of all the different views (not specifically in the sense of views in relational systems) of the underlying physical database.



The Conceptual level is concerned with transparently mapping External level views to the physical storage of the actual data in the database. In terms of a relational system, this level is composed of the database management system and the file system.

The Internal level describes the physical storage of the data on the storage media.

Detailed Architecture

The following diagram portrays the ANSI/X3/SPARC architecture in greater detail and explicitly relates the various levels to their manifestations in a relational database management system. Note the following points:

- Depending on the situation and object privileges, end users communicate with the database in one of two possible ways:
 - Indirectly through an external view
 - Directly
- Communications through views must be converted, or mapped, in both directions.
- Communications between the database and the disk subsystem are made through the file system.

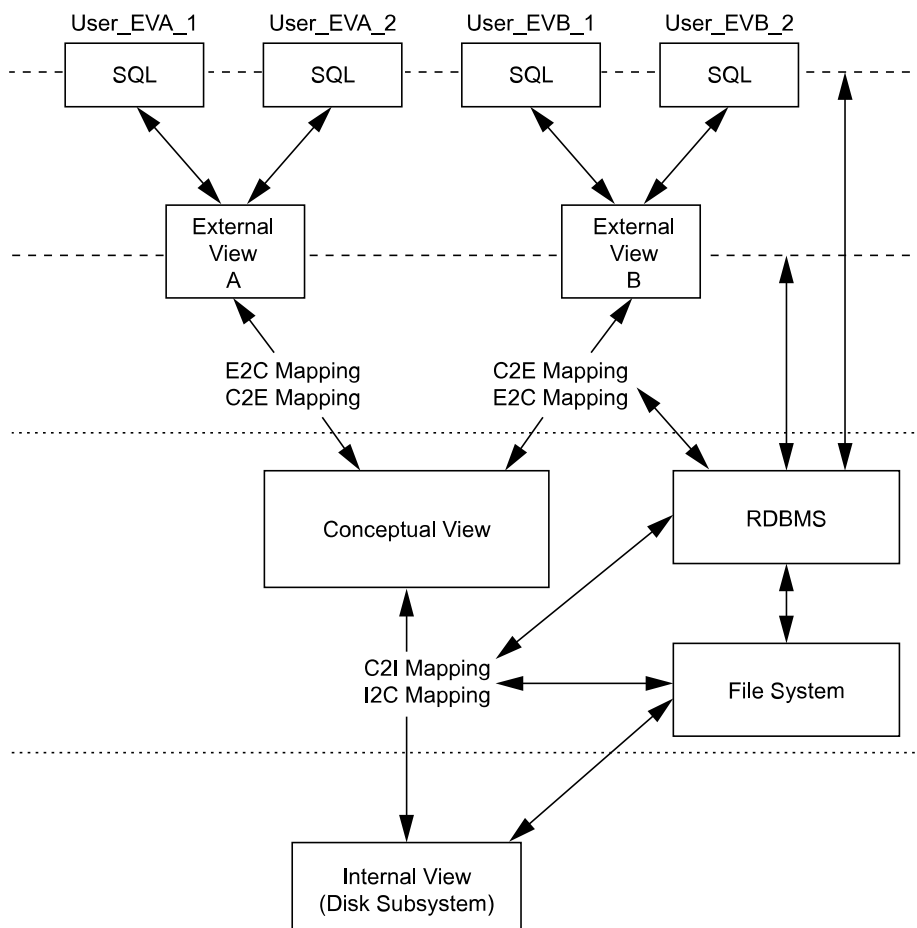
The higher level within the External level is represented by the SQL language for relational systems. The SQL language elements can be embedded within a client application, within a stored procedure, or presented to the database in the form of an ad hoc, interactive query made through a query manager, or through a natural language SQL code generator application.

The lower level within the External level is represented by a relational view. Views not only mask the underlying database storage, but also the conceptual structure it supports, acting as a virtual tabular interface on the physical base tables.

Note that not all users communicate with the database through views: some, particularly administrative users, communicate with the database directly.

The Conceptual level of the architecture is represented by the relational database management system and the file system. The role of the file system is more nebulous than the diagram portrays and is actually intermediate between the Conceptual and Internal levels, but for purposes of this description of the architecture, it is regarded as a component of the Conceptual level.

The Internal level of the architecture is represented by the disk subsystem. Depending on the configuration, the Internal level could also include storage media such as tape and optical disk. From the perspective of the architecture, only the file system has direct access to the disk subsystem, and it must map data requests and responses in a manner appropriate to the level or component it communicates with.



Key

The following table explains the abbreviations used in this detailed architecture flow diagram:

Abbreviation	Definition
EVA	External View A
EVB	External View B
E2C	External-to-Conceptual
C2E	Conceptual-to-External
C2I	Conceptual-to-Internal
I2C	Internal-to-Conceptual

Requirements Analysis

Any design process must begin with the knowledge of what is to be designed. This includes not only the proposed morphology of the end product but also the systems, policies, and procedures - the processes - of the designed product.

This fundamental knowledge is derived through a process of accumulating facts about what the eventual users of the product require to do their work in support of the enterprise.

See [Requirements Analysis](#) for more information about this process.

Logical Database Design

The requirements analysis phase of the design process reveals the real world objects and their attributes that the database must represent as well as the relationships among them.

The logical database design phase formalizes the objects, or entities, and their relationships. Another primary task of logical design is normalization to ensure, for example, that the modeled entities are modified by attributes that uniquely pertain to them.

See [Semantic Data Modeling](#) and [The Normalization Process](#) for more information about logical database design.

Activity Transaction Modeling

In this phase, which Teradata refers to as Activity Transaction Modeling, you begin to attach physical attributes to the entities, attributes, and relationships defined in the logical data model. The ATM process performs the following activities:

- Identifies the business rules of the enterprise that apply to the information to be stored in the data warehouse

- Initiates the process of identifying and defining attribute domains and constraints for physical columns
- Identifies and models database applications
- Identifies and models application transactions
- Summarizes table and join accesses by column across transactions
- Compiles a preliminary set of data demographics by computing table cardinalities, value distributions, and attaching change ratings to columns

This information is compiled and used as input to the physical design process.

For more information, see [The Activity Transaction Modeling Process](#).

Physical Database Design

Physical database design is the commitment of all the previous design stages to a physical reality.

Entities, attributes, and relationships are identified and normalized in previous phases of the design process. Attribute sets are assigned to domains.

The physical design phase identifies and creates the actual databases, base tables, constraints, indexes, partitioning, views, macros, triggers, and other objects that define the physical database that drives your data warehouse.

Much of physical design focuses on the performance aspects for the execution of queries using well-chosen indexes.

See [Indexes and Maps](#) through [Designing for Missing Information](#) for more information about physical database design.

Requirements Analysis

The principal reasons for performing a requirements analysis include:

- Getting the system right the first time
- Supporting easy user access to the system
- Producing reasonable and reliable estimates of costs
- Writing a requirements specification from the approved research information and making it available to the designer of the logical database.

The process includes, at minimum, the following tasks:

- Interviewing notable employees, both management and support staff, for information such as the following:
 - What information do they need?
 - What is the source of that information?
 - What are the tasks involved with creating and reporting the information?
 - How is the information used?
- Gathering all input screens and reports generated by the legacy system and interviewing management and support staff about what is right and wrong about these components as well as determining what sorts of new or different input and report items should be added to the new system.
- Compiling and circulating the cumulative research information you have gathered to obtain affirmation of its accuracy from all involved parties.
- Writing a requirements specification from the approved research information and making it available to the designer of the logical database.

Developing an Enterprise Data Model

An enterprise data model is a blueprint: a means of ensuring that standards for creating the IT function exist and that they are appropriately integrated with the overall function of the business enterprise the function is designed to support.

Things To Avoid When Developing an Enterprise Data Model

IT organizations often commit several common mistakes in the course of developing the architecture for their enterprise data model. The following list identifies ten of the most frequently made errors in the process of developing an enterprise architecture.

- Treating the architecture as if it were a finished product rather than a process.

Like the enterprise data warehouse, the enterprise data architecture is an ongoing process, not a fixed, concrete thing. As such, it has a continuous life cycle that supports the following areas:

- Identification of particular areas that particular decisions and technology selections affect.

- Provision of guidelines for investment decisions.
- Identification of the major processes, skills, and components needed to support the IT projects that support the enterprise.

- Assuming that technical personnel make the best architects.

Of course, an enterprise data architect must be highly technical. Architects must possess a wide ranging skill set, and that sheer technical competence in the absence of other equally important skills such as the following is a formula for failure:

- Understanding and matching business objectives and needs to technological systems
- Auditing architectural compliance to ensure its overall integrity
- Educating and consulting with users about various architectural issues and so on

- Rejecting input because it does not fit the existing (or preconceived) architecture.

Ideally, such decisions should be made by a Chief Information or Technology Officer, not by the architecture staff. The role of the architects is to keep the enterprise architecture life cycle process working, not to assume (or reject) the risks of technology decisions.

- Failing to communicate early and often.

As stated previously, the enterprise data architecture is an ongoing process that must not only be updated continuously, but also sold and re-sold to its constituents. Several effective methods for achieving this follow:

- Report all important modifications to the architecture to all affected members of the user community.
- Summarize and report case studies, both of works in progress and of completed projects, of how the architecture has been put into effect by users, including both successful and unsuccessful applications.
- Conduct periodic refresher presentations and seminars on the architecture at department or staff meetings.
- Produce architecture documentation in the form of checklists, templates, and contact lists.
- Partition the enterprise architecture want list documentation into broad sets such as must-have, should-have, and nice-to-have.
- Create an architecture intranet site or portal so users can easily access all enterprise project documentation.

- Failing to explain concepts in simple, non-technical language.

It is almost always a mistake to present an architecture project in purely technical terms. Instead, describe projects in terms of the business problems they are intended to reduce or solve.

For example, whether a user-defined function or stored procedure must be written to solve the problem is almost never relevant, nor is the question of whether the procedure should be written in SQL or C. Only the business problem to be solved is important along with a non-technical explanation of how the problem is solved by the new application.

- Mistaking standards for architecture.

Without being embedded in an enterprise data architecture, standards are nothing more than a list of things that are (or are not) permitted. For example, your standard might state that a tool for a particular category of tasks must support x, y, and z. When embedded within an enterprise architecture, those standards would also be augmented by a set of usage guidelines that map specific categories of tools to specific types of problems that the architecture is designed to solve.

- Forgetting to assess people and process impacts.

Organizations frequently evaluate technology in a void, failing to consider the impact of new technologies on people and processes. Not only are ergonomic factors often overlooked, but even such basic business components as the staffing and administration of new technologies are often not evaluated.

With respect to processes, factors such as capacity planning, security management, user administration, and operations management are also frequently forgotten.

- Aligning with strategies rather than business goals and cultural values.

The problem with this approach is that business strategies not only change frequently, but often fail. If the enterprise needs to revise its strategy, it should not also have to revise its data architecture.

Instead, consider aligning your IT architecture with the business goals and cultural values of the corporation. For example, a commonly stated business goal is to provide mid-market products at the best price with the least possible inconvenience.

An example of aligning the corporate information architecture with this goal might be optimizing supply chain performance with low cost and maximally efficient transaction processing.

- Compelling unwilling constituents to participate in architectural decisions.

Enterprise architects should seek out individuals within the organization who understand the value of a sound enterprise data architecture and are willing to participate in developing that architecture.

Avoid the temptation to coerce the participation of unwilling individuals, no matter how valuable you perceive their input to be.

- Introducing technology before its time.

Avoid introducing new technologies just because the industry perceives them to be the next big thing. Instead, focus on finding proven technologies that can address existing business problems or that can enable the delivery of new business value.

Consider the following 5 best practices for assessing the readiness of candidate technologies to support existing business problems and make it possible to bring new value into the enterprise:

- Create a technology assessment program.
- Establish pilot projects to evaluate and assess the readiness of new technologies.
- Devise training programs to enhance the skills of the staff members who will be using the new technologies.
- Design phased migration plans that parallel the evolution of updated staff member skills.
- Consider engaging in partner relationships with the technology vendors to help introduce new technologies.

Semantic Data Modeling

This section describes the basics of semantic, or logical, data modeling, focusing on entity-relationship analysis.

Teradata created several different industry-specific logical data model (LDM) frameworks, including LDMs for the following industries:

- Communications
- Financial Services
- Healthcare
- Insurance
- Manufacturing
- Media
- Retail
- Transportation and Logistics
- Travel and Hospitality Industry

Contact your Teradata representative for details about the availability of these and other LDMs.

The Entity-Relationship Model

Applications of the E-R Model

The Entity-Relationship (E-R) model is a widespread technique used for deriving an initial logical model for relational databases.

Entities, Relationships, and Attributes

Definition of an Entity

An entity is a database object that represents a thing in the real world. Entities are expressed as nouns.

Entities can be concrete, like buildings and employees or they can be more abstract things like departments and accounts.

Loosely speaking, an entity corresponds to a relation in relational theory. When a relation is made physical, it is normally referred to as a table, though the term is also used to describe physical as well as conceptual tables.

Types of Entities

There are several different schemes for categorizing entities based on qualitatively different criteria. For purposes of this document, the following two schemes are defined:

- Major and minor entities
- Supertype and subtype entities

The following table provides definitions for these types:

Entity Type	Definition	Example
Major	An entity with relatively large cardinality and degree that is updated frequently.	Order table
Minor	An entity with small cardinality and degree that is rarely updated. Minor entities are typically used in a single, 1:M association, and their primary key is often nonnumeric.	Nation Code table
Supertype	A generic entity that is a superclass of one or more subtype entities. Supertype and subtype entities model the same real world entity at a high level. Supertypes must, by definition, have one or more reciprocal subtypes.	Publications table
Subtype	A specific entity that is a disjoint subclass of one and only one supertype entity. Subtype and supertype entities model the same real world entity at a high level. Subtype entities typically have a higher degree than their supertypes, with the additional attributes describing detailed characteristics of the subtype that distinguish it from the other subtype entities of a mutual supertype.	<ul style="list-style-type: none"> • Book table • Magazine table • Professional journal table • Conference proceedings (all as subtypes of the supertype Publications)

Definition of a Relationship

A relationship is an association among two or more entities or other relationships. Relationships are expressed as verbs.

Relationships among entities are described by one of three ratios:

Relationship	Shorthand Notation
One-to-one	1:1
One-to-many	1:M
Many-to-many	M:M

Relationships as defined in the E-R model have no direct counterpart in relational theory. The closest property of relational theory to expressing what a relationship is in E-R theory is the primary key-foreign key relationship.

[Relationship Theory](#), [One-to-One Relationships](#), [One-to-Many Relationships](#), and [Many-to-Many Relationships](#) describe the properties of relationships in greater detail.

Definition of Attribute

An attribute is a characteristic of an entity. Every entity has at least one attribute: its primary key (More accurately, a *candidate* key). Attributes are expressed as nouns qualified by adjectives that clarify their role.

An attribute plays one of three possible roles in any table:

- Primary key attributes identify the entity or relationship modeled by a table.
Primary key attributes are said to be *identifier* attributes because they uniquely identify an instance of an entity.
- Foreign key attributes define relationships between and among entities or among entities and relationships.
A foreign key attribute can be an identifier attribute if it is part of a composite primary key; otherwise, foreign key attributes are descriptor attributes.
- Nonkey attributes further describe the entity or relationship modeled by a table.
Nonkey attributes are said to be *descriptor* attributes because they specify a nonunique characteristic of an instance of an entity.

In the relational model, attributes have the same properties as they do in E-R theory.

Definition of a Derivative Attribute

So-called derivative attributes violate the rules of normalization in relational theory because they are not atomic. A derivative attribute is any attribute that can be derived by calculation from other data in the model.

The issue of derivative attributes should not concern you during the logical design phase other than knowing that they should not be modeled. Derivative attributes are an important consideration for physical database design, where they are often modeled as a means for enhancing system performance.

Note that Vantage offers several features like hash and join indexes, aggregate join indexes, and global temporary tables that lessen the temptation to denormalize the physical design of your base tables by using derivative attributes.

Translating Entities and Relationships Into Tables

Definition of a Prime Table

A prime table is a table that has a single column primary key.

Prime tables always model entities and all entities are modeled by prime tables.

Ensure that all prime tables have been defined prior to defining any associative tables (see *Definition of an Associative Table* below).

The following table, Table_A, is prime because it has a simple primary key (see the definition for *Simple key* in [Definitions](#)):

Table_A
A_Key
PK
A1
A2
A3

Definition of a Non-Prime Table

A non-prime table is a table that has a composite primary key.

The following table, Table_B, is non-prime because it has a composite primary key (see the definition for *Composite key* in [Definitions](#)):

Table_B	
B_Key_1	B_Key_2
PK	
B1_1	B2_3
B1_2	B2_2
B1_3	B2_1

Definition of an Associative Table

An associative table is a non-prime table whose primary key columns are all foreign keys.

Because associative tables model pure relationships rather than entities, the rows of an associative table do not represent entities. Instead, they describe the relationships among the entities the table represents.

Always define all your prime tables before defining any associative tables.

The following associative table, table_a-b, associates prime table entity table_a with prime table entity table_b:

table_a-b	
a_Key	b_Key
PK	
FK	FK
A1	B1
A2	B5

table_a-b	
A3	B2

Guideline for Naming Associative Tables

Use the following general form to name associative tables.

```
prime_table_name_A-prime_table_name_B
```

This convention helps to keep the alphabetic and logical sequences of tables synchronized, making it easier to locate the prime tables for the foreign keys that make up its composite primary key.

Relationship Theory

Definition of Degree

The degree of a relationship is the number of entities associated in the relationship. Typical degree descriptions are provided in the following table:

For a relationship among this many entities ...	The following term is commonly used to describe the relationship ...
1	Unary
2	Binary
3	Tertiary
<i>n</i>	<i>n</i> -ary

Definition of Connectivity

Connectivity refers to the mapping of entity occurrences in a relationship. The possible values for the connectivity of a relationship are three:

- 0
- 1
- Many

Definition of Cardinality

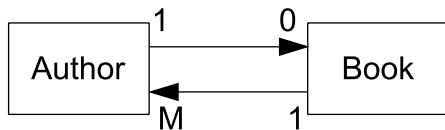
With respect to E-R theory, the term cardinality refers to the integer number represented by the symbol *M* in a 1:M or M:M relationship. This number describes the constraint on the number of entity instances that are related through the relationship.

In this context, cardinality does not refer to the number of tuples in a relation.

Definition of Existence Dependency

Whenever the existence of an entity depends on the existence of another entity, that relationship is described as an existence dependency (ED).

For example, suppose there are two entities named Author and Book. A writer might have been provided with an advance to write a book for a publisher, but if this is the first book written by the writer for this publisher, there will be no entry in the Book entity until the contracted book has been written and published. In this case, the relationship between Author and Book is 1:0, indicating that the existence of an instance of Book is optional.



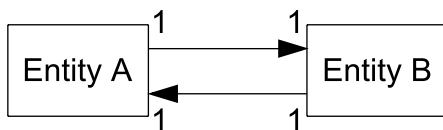
One-to-One Relationships

Assume two entities, *A* and *B*.

A 1:1 relationship exists between *A* and *B* when each occurrence of entity *A* is related to at most one occurrence of entity *B*, while each occurrence of entity *B* is related to at most one occurrence of entity *A*.

1:1 relationships are not commonly seen in real world situations.

1:1 relationships are graphed as follows:



Modeling 1:1 Relationships

1:1 relationships are modeled by placing the primary key of entity *A* as a foreign key component of entity *B* with no duplicates allowed. Because the relationship is symmetrical, you could just as well place the primary key of entity *B* as a foreign key component of entity *A*.

Guideline for Placing the Foreign Key

Place the foreign key in whichever entity minimizes or eliminates the possibility of nulls.

Example

Because of the symmetry of 1:1 relationships, the following entity pairs both model the same relationship:

A		B		A		B	
A_Key		B_Key	A_Key	A_Key	B_Key	B_Key	
PK		PK	FK, ND	PK	FK, ND	PK	
A1		B1	A1	A1	B1	B1	
A2		B2	A3	A2	B5	B2	
A3		B3	A5	A3	B2	B3	

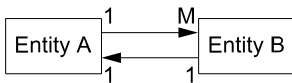
One-to-Many Relationships

Assume two entities, *A* and *B*.

A 1:M relationship exists between entity *A* and entity *B* when each occurrence of entity *A* is related to 0, 1, or more occurrences of entity *B*, while each occurrence of entity *B* is related to at most one occurrence of entity *A*.

1:M relationships are commonly seen in real world situations. For example, each department has many employees, but each employee has only one department.

1:M relationships are graphed as follows:



Modeling 1:M Relationships

1:M relationships are modeled by placing the primary key of entity *A* as a foreign key component of entity *B*.

Example

Because 1:M relationships are asymmetric, the entity *A* key must be placed in entity *B*. The reciprocal relationship does not model the same E-R relationship.

A		B	
A_Key		B_Key	A_Key
PK		PK	FK
A1		B1	A1
A2		B2	A3
A3		B3	A1
		B4	A1
		B5	A3

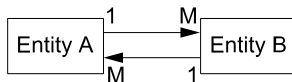
Many-to-Many Relationships

Assume two entities, *A* and *B*.

A M:M relationship exists between entity *A* and entity *B* when each occurrence of entity *A* is related 0, 1, or many occurrences of entity *B* and each occurrence of entity *B* is related to 0, 1, or many occurrences of entity *A*.

M:M relationships are commonly seen in real world situations.

M:M relationships are graphed as follows:



Modeling M:M Relationships

M:M relationships are modeled by placing the primary key of entity *A* and the primary key of entity *B* as foreign keys comprising the entire primary key of a separate entity referred to as an associative table. This is another example of the subjectivity of E-R theory, because a relationship is expressed as a physical table in the database.

Guideline for Drawing M:M Tables for Users

Draw entities that represent M:M relationships in a way that data is presented in a form most familiar to users.

Example

The following set of entities models the M:M relationship between *A* and *B*:

A	B	A-B	
A_Key	B_Key	A_Key	B_Key
PK	PK	PK	
		FK	FK
A1	B1	A1	B1
A2	B2	A2	B5
A3	B3	A3	B2

Moving From an Entity-Relationship Analysis to Normalization

The goal of an E-R analysis is to generate a reasonable set of entities and to map the relationships among them.

The next step in the process of achieving a logical data model for your database is to translate those entities into a set of fully normalized tables.

The topics of normal forms and the normalization process are described in [The Normalization Process](#).

The Normalization Process

This section reviews some concepts of the normalization process. It assumes that any designer will use a CASE tool that has its own set of notational structures and conventions, so the information is as generic as possible.

Note that many of the dependencies described in this section can also be used to optimize SQL requests (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142).

Vantage promotes full database normalization for logical modeling. The reasons for normalized databases include the following factors:

- Normalization is a provable, logical, and consistent method for designing provably correct database schemas.
- The Teradata parallel architecture was designed from the outset to support normalized databases.

The normalization process helps you to structure your thinking about the entities you have identified and the relationships they share among one another. It heightens your awareness of the problems that can occur when all the attributes of a schema are not sorted out and stored in one and only one place.

Because database management systems map logical relations directly to physical tables, it can sometimes appear difficult to separate the logical model from its physical realization. Nevertheless, you should always design a normalized logical model, then, only if necessary to achieve performance levels that are otherwise not possible to realize, *denormalize* the physical design.

There can be a cost to normalization when the logical model is implemented as a 1-to-1 physical mapping. In a very real sense, normalization optimizes update performance at the expense of retrieval performance. Because of this, it sometimes becomes necessary to denormalize physical tables to some extent in order to achieve reasonable overall system performance. Denormalization is actually an implementation issue, not a logical design issue.

This document often refers to denormalization in the context of physical database design because the logical and physical models of a database are independent things. The physical model applies only to implementation. See [Denormalizing the Physical Schema](#) for a description of denormalization.

Before examining the details of the normalization process, you should understand the properties of relations and their logical manipulation described in [Properties of Relations and Their Logical Manipulation](#).

Properties of Relations and Their Logical Manipulation

Definitions

Term	Definition
<i>Alternate key</i>	Any candidate key not selected to be the primary key for a relation.
<i>Attribute</i>	A property of a relation that describes its primary key. Each attribute has a unique name, is drawn from a domain, and can be constrained in various ways.

Term	Definition
	<p>Because its values are all drawn from the same domain, the data for any attribute is homogeneous by definition.</p> <p>Attribute is the term used in set theory and logical design. Column is the equivalent term used in physical design and database management.</p>
<i>Body</i>	<p>The body of a relation is the composite value set assigned to its tuple variables.</p> <p>Each SQL relation must have a body (see Types of Missing Values about "Table_Dee" and "Table_Dum" for why this might not be an optimal situation).</p>
<i>Candidate key</i>	<p>An attribute set that uniquely identifies a tuple.</p> <p>A candidate key has the following minimal properties:</p> <ul style="list-style-type: none"> • The value of the key uniquely identifies the tuple in which it appears. • Attributes defining the candidate key cannot be redundant. If an attribute is removed from the candidate key, then its uniqueness must be destroyed else it is not a properly defined candidate key. Compare with <i>Superkey</i>. <p>In the completed physical design of a relational database, candidate keys are easy to identify because they are always constrained as UNIQUE NOT NULL.</p>
<i>Composite key</i>	A key defined on more than one attribute. Compare with <i>Simple key</i> .
<i>Domain</i>	<p>The set of all possible values that can be specified for a given attribute.</p> <p>The physical representation of a domain is a data type, and the ideal representation of a domain is a distinct user-defined data type (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 and <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147 for more information about UDTs).</p>
<i>Field</i>	<p>The intersection of a tuple and an attribute.</p> <p>Columns in a table are often referred to as fields, but strictly speaking, that is incorrect.</p>
<i>Foreign key</i>	<p>An attribute set based on an identical attribute set that acts as a candidate key (typically the primary key) for a different relation.</p> <p>Foreign keys reflect relationships between tables and are often used as join columns.</p>
<i>Heading</i>	<p>Each attribute of a relation must have a heading. Each such attribute has two required parts: a name and a domain, or data type. It is common practice not to call out the domain of an attribute unless it is germane to the problem at hand, but that does not render the typing of each attribute in a relation any less necessary.</p>
<i>Instance</i>	A tuple drawn from the complete set of tuples for a relation. The term is sometimes used to describe any selected set of tuples from a relation.
<i>Intelligent key</i>	<p>An overloaded simple key that encodes more than one fact (a combination of identification as well as characterization facts.), which is a violation of 1NF. The principal implementation problem with intelligent keys is that if any of the components of the key change, then all applications that access the key are affected. This is why you should always select unchanging attributes for your keys.</p> <p>The classic example of an intelligent key is the International Standard Book Number (ISBN) used by publishers to identify individual books. Each ISBN is composed of a group identifier, a publisher identifier, a title identifier, and a check digit.</p>
<i>Key</i>	An attribute set that uniquely identifies each tuple in a relation. Implicitly synonymous with primary key, though it applies equally well to any candidate or foreign key. See <i>Primary key</i> .

Term	Definition
<i>Natural key</i>	<p>The representation of a real world tuple identifier in a relational database. For example, a common identifier of employees in a corporation is a unique employee number. An employee is assigned an employee number whether that information is stored within the database or not.</p> <p>Natural keys are sometimes confused with intelligent keys (see <i>Intelligent key</i>), but they are very different concepts.</p>
<i>Order independence</i>	<p>The logical ordering of tuples and attributes has no meaning. In other words:</p> <ul style="list-style-type: none"> • Tuples have no essential up-down order. • Attributes have no essential right-left order. <p>When attributes are manifested physically as columns, their order does have significance for SQL.</p>
<i>Primary key</i>	<p>The primary key for a relation is an attribute set that uniquely identifies each tuple in the relation. By the entity integrity rule, primary keys cannot contain nulls. See Rules for Primary Keys for an explanation of the entity integrity rule.</p> <p>Every relation must have one and only one primary key. More current thinking argues that every relation must have a <i>candidate</i> key, but that it need not necessarily be declared as the <i>primary</i> key for the relation.</p> <p>Note that you cannot use XML, BLOB or CLOB columns to define a physical key or other database constraint (see Designing for Database Integrity), nor can you use XML, BLOB, or CLOB columns to define the physical primary key for a global temporary trace table. See CREATE GLOBAL TEMPORARY TRACE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i>, B035-1184</p> <p>The primary key is often used as the primary index for a table when a relation is manifested physically (see Indexes and Maps and Primary Index, Primary AMP Index, and NoPI Objects).</p>
<i>Relation</i>	<p>A two-dimensional representation of data in tabular form. Note that database relations are <i>n</i>-dimensional, not 2-dimensional as is commonly asserted.</p> <p>Relation is the term used in set theory and logical design. Table is the analogous term used in physical design and database management.</p>
<i>Relational schema</i>	A set of relations in a logical relational model. In the physical model, a relational schema is manifested as a database.
<i>Repeating group</i>	A collection of logically related attributes that occur more than once in a tuple.
<i>Simple key</i>	A key defined on a single attribute. Compare with <i>Composite key</i> .
<i>Superkey</i>	Any set of attributes that uniquely identifies a tuple, whether redundantly or not. The allowance of redundant attributes within a super key distinguishes it from a simple candidate key (see <i>Candidate key</i>).
<i>Surrogate key</i>	<p>An artificial simple key used to identify individual entities when there is no natural key or when the situation demands a noncomposite key, but no natural noncomposite key exists. Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among them. You should only resort to surrogate keys if there is no other way to uniquely identify the rows of a table.</p> <p>Surrogate keys are typically arbitrary system-generated sequential integers. See CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i>, B035-1184 for information about how to generate surrogate keys in Teradata.</p>

Term	Definition
<i>Tuple</i>	<p>A unique instance of a relation consisting of, at minimum, a primary key and zero or more attributes that describe the primary key.</p> <p>Tuple is the term used in set theory and logical design. Row is the equivalent term used in physical design and database management. Nonrelational systems use the term record in the same way relational systems use row.</p>
<i>Uniqueness</i>	<p>Duplicate rows are not permitted.</p> <p>When relations are manifested physically as tables in SQL databases, duplicate rows are permitted for multiset tables only. In fact, the ANSI/ISO SQL standard is bag (multiset) -oriented rather than set-oriented by definition. This is at odds with one of the fundamental properties of the relational model.</p> <p>The term <i>duplicate row</i> is used here as a row whose columnar values match every columnar value of one or more other rows in a relation.</p> <p>This is not a recommended practice: you should always either define your tables as set tables to avoid the many problems that multiset tables present or define them as multiset tables, but specify at least one of the columns to be UNIQUE NOT NULL.</p>

Logical Operations on Relations

Relations are formally decomposed and constructed using logical relational operators and the relational algebra. The relational algebra is a set of procedural constructs for manipulating relations, while the relational calculus is a set of nonprocedural constructs for performing the same operations, SQL is a combination of both the algebra and the calculus.

The following table lists set theory operators.

Set Theory Operators

Logical Operator	Description
DIFFERENCE	<p>$X - Y$</p> <p>The set of all attributes contained in relation X but not in relation Y.</p>
INTERSECTION	<p>$X \cap Y$</p> <p>The set of all attributes contained in both relation X and relation Y.</p>
PRODUCT	<p>$X \cdot Y$</p> <p>The set of all multiples of all attributes contained in relations X and Y.</p>
UNION	<p>$X \cup Y$</p> <p>The set of all attributes contained in either relation X or relation Y or both.</p>

Special Relational Database Operators

Logical Operator	Description
DIVIDE	<p>The division of relation R of degree $m + n$ by relation S of degree n produces a quotient relation of degree m.</p>

Logical Operator	Description
JOIN	The join of relation R on attribute X with relation S on attribute Y is the set of all tuples <i>t</i> such that the concatenation of a tuple <i>r</i> , belonging to R , and a tuple <i>s</i> , belonging to S , and the predicate <i>r.R equality_operator s.S</i> evaluates to TRUE. In SQL, this is expressed in an ON clause in a DML join statement.
PROJECT	The projection of relation R on attributes X, Y, ..., <i>n</i> is the set of all tuples (<i>x</i> , <i>y</i> , ..., <i>n</i>) such that a tuple <i>n</i> appears in R with X value <i>x</i> , Y value <i>y</i> , and so on. Less formally, a projection on relation R is any subset of the attributes of R . In SQL, this is expressed as a column list in a DML statement.
RESTRICT/ SELECT	The restriction of relation R is the set of all tuples <i>t</i> such that the comparison <i>t.X operator t.Y</i> evaluates to TRUE. Less formally, a restriction (or selection) on relation R is any subset of the tuples of R satisfying the condition <i>X equality_operator Y</i> . In SQL, this is expressed in a WHERE clause in a DML statement.

Functional, Transitive, and Multivalued Compatibilities

Definitions

Term	Definition
Determinant	For any relation R , the set of attributes X in the functional dependency $X \rightarrow Y$ is the determinant group for the dependency.
Full functional dependence	An attribute set X is fully functionally dependent on another attribute set Y if X is functionally dependent on the whole set of Y, but not on any subset of Y.
Functional dependence	For any relation R , attribute X is functionally dependent on attribute Y if for every valid instance, the value of Y determines the value of X. The symbolic notation for this is $Y \rightarrow X$.
Multivalued dependence	For any relation R with attribute set (X,Y,Z), the set X multivalue determines the set Y if, for every pair of tuples containing duplicates in X, the instance also contains the pair of tuples obtained by interchanging the Y tuples in the original pair. The symbolic notation for this is $X \rightrightarrows Y$.
Transitive dependence	Transitive dependencies are dependencies that exist among three or more attributes in a relation in such a way that one attribute determines a third by way of an intermediate attribute. For example, consider the relation R (X,Y,Z), where X is the primary key. Attribute Z is transitively dependent on attribute X if attribute Y satisfies the following dependencies, where the symbol \nrightarrow indicates "does not determine": <ul style="list-style-type: none"> $X \nrightarrow Y$ $Y \nrightarrow Z$ $Y \rightarrow X$

Inference Axioms for Functional Dependencies

The concept of functional dependencies for relations produced a complete set of inference axioms for functional and multivalued dependencies for relations. These axioms are described in the following table:

Axiom	Formal Expression					
Reflexive rule	$X \rightarrow X$					
Augmentation rule	IF	$X \rightarrow Y$	THEN	$XZ \rightarrow Y$		
Union rule	IF	$X \rightarrow Y$	AND	$X \rightarrow Z$	THEN	$X \rightarrow YZ$
Decomposition rule	IF	$X \rightarrow Y$	THEN	$X \rightarrow Z$	WHERE	Z is a subset of Y
Transitivity rule	IF	$X \rightarrow Y$	AND	$Y \rightarrow Z$	THEN	$X \rightarrow Z$
Pseudotransitivity rule	IF	$X \rightarrow Y$	AND	$YZ \rightarrow W$	THEN	$XZ \rightarrow W$

For the sake of some examples, assume the attribute set R, S, T, U, V with the following set of dependencies:

- $R \rightarrow S$
- $TU \rightarrow R$
- $T \rightarrow V$
- $V \rightarrow T$
- $SU \rightarrow T$

The following table lists many of the dependencies among these attributes implied by the inference axioms:

Dependency	Supporting Axiom
$R \rightarrow R$	Reflexive rule
$RT \rightarrow S$	Augmentation rule
$TU \rightarrow RV$	<ul style="list-style-type: none"> • Augmentation rule • Union rule
$RU \rightarrow T$	Pseudotransitivity rule
$UV \rightarrow R$	Pseudotransitivity rule
$TU \rightarrow S$	Transitivity rule
$SU \rightarrow V$	Transitivity rule
$VU \rightarrow R$	Pseudotransitivity rule
$RU \rightarrow V$	<ul style="list-style-type: none"> • Transitivity rule • Pseudotransitivity rule
$UV \rightarrow S$	<ul style="list-style-type: none"> • Transitivity rule • Pseudotransitivity rule

Inference Axioms for Multivalued Dependencies

The following table lists the complete set of multivalued dependencies:

Axiom	Formal Expression							
Reflexive rule	$X \Rightarrow X$							
Augmentation rule	IF	$X \Rightarrow Y$	THEN	$XZ \Rightarrow Y$				
Union rule	IF	$X \Rightarrow Y$	AND	$X \Rightarrow Z$	THEN	$X \Rightarrow YZ$		
Decomposition rule	IF	$X \Rightarrow Y$	AND	$X \Rightarrow Z$	THEN	$X \Rightarrow Y \cap Z$	AND	$X \Rightarrow (Z - Y)$
Transitivity rule	IF	$X \Rightarrow Y$	AND	$Y \Rightarrow Z$	THEN	$X \Rightarrow (Z - Y)$		
Pseudotransitivity rule	IF	$X \Rightarrow Y$	AND	$YW \Rightarrow Z$	THEN	$XW \Rightarrow (Z - YW)$		
Complement rule	IF	$X \Rightarrow Y$	AND	$Z \Rightarrow R - XY$	THEN	$X \Rightarrow Z$		
Mixed inference rules	IF	$X \rightarrow Y$	THEN	$X \Rightarrow Y$				
	IF	$X \Rightarrow Y$	AND	$Z \Rightarrow W$ WHERE W is contained in Y AND $Y \cap Z$ is not empty			THEN	$X \rightarrow W$

The Normal Forms

Normalization theory is constructed around normal forms that define a system of constraints. If the form of a relation meets the constraints of a particular normal form, it is said to be in that form.

While there many normal forms, the third normal form (3NF) discussed in this section is adequate for most logical database designs.

The Objective of Normalization

The intent of normalizing a relational database can be reduced to one simple aphorism: One Fact In One Place. By decomposing your relations into fully normalized forms, you can eliminate the majority of update anomalies that can occur when data is stored in unnormalized tables. Decomposition is attained through a series of projections of a relational schema into a set of normalized relations that optimize the conciseness of attributes.

A slightly more detailed statement of this principle would be the definition of a relation (or table) in a normalized relational database: A relation consists of a primary key (more accurately, a *candidate* key.), which uniquely identifies any tuple, and zero or more additional attributes, each of which represents a single-valued (atomic) property of the entity type identified by the primary key (once again, it is more accurate to say *candidate key* in place of primary key).

Types of Decomposition

Relations can be decomposed in one of two ways: horizontally or vertically.

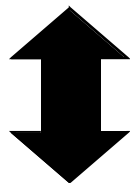
A horizontal decomposition is one in which a relation is partitioned along its cardinality dimension. In other words, entire tuples are divided into two or more tuple sets. Conceptually, this is pure relational restriction without projection, as indicated by the following graphic:

Supplier_Parts

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105
23	Albertine	P-063	10
84	Lê	P-346	622

Supplier_Parts_1

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105



Supplier_Parts_2

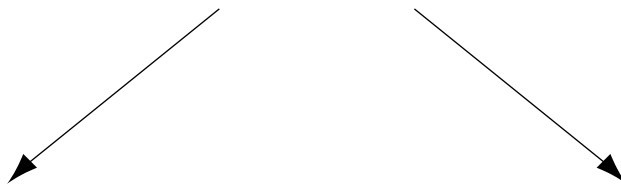
SuppNum	SuppName	PartNum	Quantity
23	Albertine	P-063	10
84	Lê	P-346	622

A vertical decomposition is one in which a relation is partitioned along its arity dimension. In other words, the attributes of a relation are decomposed into two or more sets of projections.

Conceptually, this is pure relational projection without restriction, as indicated by the following graphic:

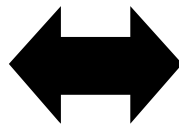
Supplier_Parts

SuppNum	SuppName	PartNum	Quantity
1	Gozzo	P-001	251
1	Gozzo	P-002	719
17	Nagata	P-942	105
23	Albertine	P-063	10
84	Lê	P-346	622



Supplier

SuppNum	SuppName
1	Gozzo
1	Gozzo
17	Nagata
23	Albertine
84	Lê



Parts

SuppNum	PartNum	Quantity
1	P-001	251
1	P-002	719
17	P-942	105
23	P-063	10
84	P-346	622

Notice that the vertical decomposition in this example is not a simple splitting of attributes between relvars (see [Relations, Relation Values, and Relation Variables](#)), but an actual normalization step (to BCNF), so the *SuppNum* attribute is duplicated, once as a PK in Supplier and again as an FK in Parts.

A more pure example of vertical composition is the projection of a subset of the columns of a large table into a single-table join index (see [Single-Table Join Indexes](#)).

For more information about decomposing relations, see [Decomposing Relations](#).

Third and Boyce-Codd Normal Forms

Third Normal Form (3NF) deals with the elimination of nonkey attributes that do not describe the primary key (more accurately, 3NF deals with the elimination of nonkey attributes that do not describe any *candidate* key for the relation).

Definition of Third Normal Form

A relation variable R is said to be in Third Normal Form when it is in 2NF and one of the following statements is also true for every nontrivial functional dependency (assume the functional dependency $X \rightarrow A$):

- The attribute set x is a superkey.
- Attribute A is part of some candidate key.

Another way of stating the rule is this: The relationship between any two nonkey attributes or attribute sets (excluding attributes having *no duplicates allowed* constraints) must not be one-to-one in either direction.

This condition is met when both of the following criteria have been met:

- Every nonkey attribute depends on all attributes of the primary key: the entire primary key.
Again, this is more accurately stated as the entire *candidate* key.
- No non-key attribute is functionally dependent on another non-key attribute of the relation.

The formal definition for Third Normal Form is as follows: For a relation to be in 3NF, the relationship between any two nonprimary key attributes (more accurately, between any two non-candidate key attributes) or groups of attributes in a relation must not be one-to-one in either direction. In other words, the nonkey attributes are nontransitively dependent upon each other and the key. Having no transitive dependencies in a relation implies no mutual dependencies.

Attributes are said to be mutually independent if none of them is functionally dependent on any combination of the others. This mutual independence ensures that individual attributes can be updated without any danger of affecting any other attribute in a tuple.

This is an incomplete definition for 3NF that fails to account for functional or transitive dependencies. To account for this special case, the definition of 3NF is extended as described in the following section.

Definition of Boyce-Codd Normal Form

When the relational model of database management was originally proposed, it only addressed what are now known to be the first three normal forms. Later theoretical work with the model showed that 3NF required further refinement to eliminate certain update anomalies.

The classic definition for third normal form does not handle situations in which a relation R has multiple composite candidate keys with overlapping attributes such as the following, where CK_1 , CK_2 , and CK_3 represent overlapping candidate keys on the overlapping attribute composites A_1-A_2 , A_2-A_3 , and A_3-A_4 , respectively:

Relation R

A ₁	A ₂	A ₃	A ₄	A ₅	A ₆
CK ₁		CK ₃			
	CK ₂				

While this situation does not occur frequently, it does present itself from time to time.

Boyce-Codd Normal Form addresses this situation. BCNF reduces to 3NF whenever the special situation that defines this problem does not apply.

A relation is in BCNF if and only if every determinant is a candidate key. This means that only determinants can be candidate keys.

Somewhat more formally, a relation is said to be in BCNF when it is in 2NF and the following is true: if whenever $X \rightarrow A$ and A does not belong to X , then X is a superkey.

Meaning of Third Normal Form in This Document

For purposes of this document, use of the term 3NF generally covers BCNF as well. Relations in BCNF are often awkward to deal with, however, and are usually more tractable if decomposed into other relations in 4NF.

Example: 3NF

For example, consider the following relations:

Customer

CustNum	CustName
PK	
1	Wright
2	Adams

Order

OrderNum	OrderDate	CustNum
PK		FK
1	2000/03/15	1
2	2000/03/17	2
3	2000/04/15	1

If these relations had been designed so that the *CustName* attribute was in the *Order* relation, 3NF would have been violated because there would be a one-to-one relationship between two nonkey attributes, *CustNum* and *CustName*.

Example: Violating 3NF

Suppose the *Order* relation had been structured like this:

Order			
OrderNum	OrderDate	CustomerNum	CustomerName
PK		FK	
1	2000/01/15	1	Wright
2	2000/02/17	2	Adams
3	2000/02/01	3	Wright

The potential update anomalies associated with this violation of 3NF are the following:

- Changed customer name.

To change the name of a customer, you must find every occurrence of its customer name.

- Inconsistent customer names.

Adams might be misspelled "Addams" in an occurrence, which would cause the tuple containing the misspelled customer name to be missed in a query having the WHERE predicate `customername = Adams`.

Worse still, customer number 2 might be labeled Adams in one tuple and Zoller in another.

- Inability to add new customer names unless they have an order placed.

Example: BCNF

Now consider the following entity, which is in 3NF, but still has some problems:

Schedule				
Campus	Class	Section	Date	Building/Room
PK				
Rancho Bernardo	Physical Database Design	1	2000/05/15	A/225
El Segundo	Relational Database Modeling Workshop	2	2000/08/15	ES/16-201
Dayton	Vantage Basics	3	2000/06/30	CTEC/120
Rancho Bernardo	Advanced SQL for Vantage	1	2000/06/30	A/225

Note that the buildings determine campus because no two buildings on any of the Teradata campuses have the same abbreviation. As a result, Building/Room \rightarrow Campus. The relation is not in Boyce-Codd normal form.

Normalize the relation by decomposing it into the two following BCNF relations:

Campus-Class

Campus	Class	Section	Date
PK			
Rancho Bernardo	Physical Database Design	1	2000/05/15
El Segundo	Relational Database Modeling Workshop	2	2000/08/15
Dayton	Vantage Basics	3	2000/06/30
Rancho Bernardo	Advanced SQL for Vantage	1	2000/11/23

Building-Campus

Building/Room	Campus
PK	
A/225	Rancho Bernardo
ES/16-201	El Segundo
CTEC/120	Dayton

Decomposing Relations

The principal goal of normalization is to eliminate update anomalies. By decomposing your relations into normalized forms, you can eliminate the vast majority of update anomalies.

Advocates of dimensional modeling argue against the method of decomposition because normalization tends to produce many relations. This is an implementation issue, not a logical design issue, and it originates in the inability of most vendor database products to make reasonably high-performing joins.

When you design enterprise database schemas using the dimensional model (see [Dimensional Modeling, Star, and Snowflake Schemas](#)), you create only a few fact relations with a composite primary key that is not free of transitive dependencies (see [Functional, Transitive, and Multivalued Compatibilities](#)), and smaller satellite relations called dimensions that contain detailed data about the fact relation.

An enterprise schema design using the dimensional model (see [Dimensional Modeling, Star, and Snowflake Schemas](#)) creates only a few fact relations with a composite primary key that is not free of transitive dependencies (see [Functional, Transitive, and Multivalued Compatibilities](#)), and smaller satellite relations called dimensions that contain detailed data about the fact relation.

Decomposing Relations for an Inherently Parallel Database Environment

Because the Vantage architecture is inherently parallel and optimized for join performance, it suffers no performance deficits when dealing with a physical schema that has been mapped directly from a normalized logical design.

Identifying Candidate Primary Keys

The primary key for a relation variable is an attribute set that uniquely identifies each tuple in that relation variable. This property is true for any alternate key, not just the candidate key that is selected to be the primary key for a relation.

Primary keys do not denote either of the following properties.

- Order
Tuples within a relation are not ordered in any way.
- Access path
Keep in mind that normalization is a logical process, not a physical implementation of the database. Primary keys are not used to access rows on disk, though they are a frequent choice as the primary index for tables (see [Indexes and Maps](#) and [Primary Index, Primary AMP Index, and NoPI Objects](#)). The primary index does define a storage path and at least one access path for table rows.

Procedure for Identifying Primary Keys

The selection of a primary key for a relation is the final step in a (possible) series of identifications of unique attribute sets as follows:

1. Identify any superkeys that exist in the attribute set.
A superkey is any set of (possibly redundant) attributes that uniquely identifies the tuples of a relation. Every relation has at least one superkey and might have only one.
2. Eliminate any redundant, or otherwise unnecessary, attributes in the identified superkeys and produce a candidate key set.
A candidate key is a nonredundant attribute set that uniquely identifies the tuples of a relation. Note that it is possible for composite candidate keys to overlap.
By definition, every relation has at least one candidate key and often has only one candidate key.
3. Select the primary key from the set of identified candidate keys.
Selection of the primary key from a set of candidate keys is ultimately an arbitrary decision, but when there are multiple candidate keys to choose from, a good rule of thumb is to select the one having the fewest attributes, particularly if you plan to use the primary key as the primary index for the table.
See [Performance Considerations for Primary Indexes](#) for specific information on selecting primary indexes to optimize retrieval and hashing performance.

The unselected candidate keys, if any, are referred to as alternate keys. Assign a UNIQUE constraint to any alternate key not selected to be the primary key for a relation to ensure its integrity with respect to the *referential integrity rule* (see [The Referential Integrity Rule](#)).

Every relation has one and only one primary key.

Surrogate Keys

Situations occur where the identification and choice of a simple primary key is difficult, if not impossible. There might be no single column that uniquely identifies the tuples of a relation variable or, looking ahead to physical design, there might be performance or query condition considerations that argue against using a composite key. In these situations, and only in these situations, surrogate keys are a solution.

As previously defined (see *Surrogate key* in [Definitions](#)), a surrogate key is an artificial, simple key used to identify individual entities when there is no natural key or when the situation demands a simple key, but no natural simple key exists.

Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among those entities. As a result, it is often difficult to maintain referential integrity relationships among tables with surrogate keys.

Surrogate keys are typically arbitrary system-generated sequential integers. See the information about CREATE TABLE (Column Definition Clause) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for information about how to generate surrogate keys in Vantage.

Rules for Primary Keys

There are several rules that define the bounds of primary keys. Note that these rules actually apply to all *candidate* keys for a relation, not just to its primary key.

The first and second rules in the following list are absolute, and the third is strongly advised:

- Primary key attributes cannot be null.

This principle is known as the *entity integrity rule*, and it is one of the fundamental principles of relational database theory. See [The Referential Integrity Rule](#) for a definition of the other fundamental integrity rule.

By definition, nulls are not unique because they represent missing values that cannot be distinguished from one another.

Although the database relational model does not explicitly state that alternate keys cannot be null, the constraint is implicit because a column set cannot be a potential primary key (a candidate key) if it is null or contains nulls.

- Primary and alternate key attributes cannot contain duplicate values.

By definition, a primary key is a unique identifier. If it contains duplicate values, it cannot be unique (this means that multiset tables cannot have true primary keys).

- Primary and alternate key values should never be modified.

This rule is neither part of the relational model, nor is it absolute, because there are occasions when primary key updates must be made.

Whenever a primary key is updated, it is possible, and even probable, that a series of coordinated foreign key updates must also be made to maintain the consistency of the database. However, it is equally likely that those cascaded updates will never be performed, leaving the database in a state that does not reflect reality even if all the system integrity constraints have been satisfied, which is the principal reason primary key updates are discouraged so strongly.

Guidelines for Selecting Primary Keys

This topic presents several recommendations for selecting the attributes that make up the primary key for a relation variable. These guidelines apply, the necessary changes being made, to all candidate keys:

- Select numeric attributes as primary keys.

Numeric keys are both easier to produce uniquely and easier to manage than character data.

Always assign numeric attributes as the primary key if the key is to be system-generated.

- Whenever possible, use system-assigned keys (see *Surrogate key* in [Definitions](#)) to simplify the maintenance of uniqueness on the primary key attribute set. This is another factor that argues in favor of numeric keys.
- Select primary key attributes that can remain unique for the life of a relation variable.
- Construct primary keys on attributes that rarely, if ever, change during the life of a relation variable.
- Never use intelligent keys.

An intelligent key carries semantics about the tuple it identifies.

The expression is also used to describe a key constructed around one or more misprojected attributes that would not be in the relation variable were the database properly normalized.

Note that intelligent keys and natural keys are not the same thing. See *Intelligent key* and *Natural key* in [Definitions](#) for details.

- Whenever relation variables are paired in a supertype-subtype relationship, always assign the same primary key to both.
- Use a consistent convention when naming primary key attributes (see below).

Because it is important to track the originating entity for attributes, always identify foreign keys with a unique code.

For example, if you use the column naming convention described in [Guidelines for Naming Columns](#), you might extend the convention as indicated in the following example:

Suppose the relation *Lineitem* has the following primary key composed of the three attributes indicated by preliminary attribute names:

Line_item		
line_item_number	order_number	order_year_date
PK		
	FK	FK

You might consider naming the primary key attributes as follows.

- line_item_number
- line_item_order_FK_number
- line_item_order_FK_year_date

The FK embedded in order_number and order_year_date indicate that the attributes are foreign keys that reference the *order* relation variable.

When attribute naming has been completed, the primary key attribute names should look like the following relation definition fragment:

Line_item with Renamed Primary Key Attributes		
line_item_number	order_FK_number	order_FK_year_date
PK		
	FK	FK

Foreign Keys

A foreign key is an attribute set in one relation based on an identical attribute set that acts as a primary or alternate key in a different relation.

Foreign keys are a special form of [Inclusion Compatibilities](#).

Foreign keys provide a mechanism for performing primary index joins, sometimes referred to as prime key joins.

Foreign keys are also used to maintain referential integrity among related tables in a database (see [The Referential Integrity Rule](#)).

Rules

- Foreign key values are restricted to three types.
 - A mirror image, drawn from the same domain, of a primary or alternate key in an associated relation.
 - Wholly null
 - Partially null

The partially null case applies to compound foreign keys only, where one or more columns of the key might be null while others might contain references to primary keys in other tables, which are, by definition, non-null.

Because of the myriad problems nulls present in database management (see [Designing for Missing Information](#)), you should avoid creating foreign keys that are either wholly or partially null.

- You cannot use BLOB or CLOB columns to define a physical foreign key or other database constraint (see [Designing for Database Integrity](#)).

- You cannot define a foreign key for a global temporary trace table. See the information about CREATE GLOBAL TEMPORARY TRACE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

The Referential Integrity Rule

The intent of the referential integrity rule is to ensure that if some tuple r in relation variable R references some tuple s in relation variable S , then tuple s must exist.

This prevents you from corrupting the integrity of the database by deleting data that is still being used by another relation.

If a base relation R has a foreign key FK that matches the primary key PK of base relation S , then every value of FK in R must have one of the following properties:

- Equal to the value PK of some tuple in S .
- Wholly null (each value in the attribute set that defines FK is null).

This definition presents an obvious contradiction and is not supported by relational theory. By definition, a candidate key (and the FK in a referential relationship must be a candidate key for its relation) must be a unique identifier for its relation, yet a key that is wholly null contains no values, only markers for values that are missing; therefore, it cannot be unique.

If your enterprise business rules prohibit nulls in foreign key columns, you can enforce a non-null constraint on a column by defining the column with a NOT NULL attribute using either CREATE TABLE or ALTER TABLE.

In other words, a row cannot exist in a table with a (non-null) value for a referencing column if no equal value exists in its referenced column.

This relationship is referred to as referential integrity and the act of ensuring that this rule is enforced is referred to as maintaining referential integrity.

This principle is known as the *referential integrity rule*, and it is one of the fundamental principles of relational database theory. Besides ensuring database integrity, referential integrity constraints have the added benefit that they are frequently used to optimize join plans when implemented in your physical database design.

The referential integrity rule applies equally, the necessary changes being made, to all candidate keys, not just the key selected to be the primary key for a relation.

Entity Integrity and Foreign Key Nulls

Permitting a foreign key to be wholly null seems to contradict the entity integrity rule (see [Rules for Primary Keys](#)). This property is, to be sure, somewhat ad hoc, but it can have some practical usefulness. For example, consider the following case: an employee is working for some enterprise but has not yet been assigned to a department. Assuming that *Department Number* is a FK in the *Employee* relation, you can readily see that even when all the remaining information required to create an *Employee* relation tuple for the employee in question is available, you cannot create such a tuple in the *Employee* relation unless you also allow the FK to be null.

The tuple in question should be updated with the appropriate *Department Number* data just as soon as it is available, but allowing the null FK permits the enterprise to pay this employee even though she is not yet assigned to a department.

Alternatively, the problem can be avoided entirely with a minor change in the logical design of the database. See [Redesigning the Database to Eliminate the Need for Nulls](#) for an example.

As another alternative, you can assign default values to the FK field set and then update with the appropriate *Department Number* as soon as you know what it is. SQL facilities for assigning default values to columns are described in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Enforcing the Referential Integrity Rule

The relational model states the Referential Integrity rule without providing a mechanism to enforce it. In practice, the rule is enforced through the PRIMARY KEY and FOREIGN KEY clauses in the CREATE TABLE statement. The keyword REFERENCES describes the table and column set in which the primary key image of the foreign key resides.

The REFERENCES WITH NO CHECK OPTION specification explicitly instructs Vantage not to enforce referential integrity on the specified relationship. See [Semantic Constraint Specifications](#) for further information.

When referential integrity is enforced, you cannot delete a row from the referenced (parent) table as long as there is a row in the referencing table whose foreign key matches it.

Enforcing Referential Integrity

To implement referential integrity (RI) in Vantage, you have three choices, ranked in their order of preference:

1. Use the declarative referential integrity constraint checks supplied and enforced by the database software.
2. Write your own, site-specific macros, triggers, or stored procedures to enforce RI.
3. Enforce constraints through application code.

See the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for a description of the problems inherent in attempting to enforce database integrity using nondeclarative constraints.

Definition of a Referencing (Child) Table

The referencing table is referred to as the Child table, and the specified Child table columns are called *referencing columns*.

Referencing columns should be of the same number and have the same data type as the referenced table key.

Definition of a Referenced (Parent) Table

A child must have a parent, and the referenced table in a PK-FK relationship is referred to as the Parent table. The key columns in the Parent table that see a Child table are called *referenced columns*.

Since the referenced columns are defined as unique constraints, the referenced column set must be one of the following:

- A unique primary index (UPI), not null
- A unique secondary index (USI), not null

Definition of a Primary Key

With respect to RI, a primary key (more accurately, a *candidate* key) is a parent table column set that is referred to by a foreign key column set in a child table.

Definition of a Foreign Key

With respect to RI, a foreign key is a child table column set that refers to a primary key column set in a parent table.

Why Referential Integrity Is Important

Referential integrity is a mechanism to keep you from corrupting your database. Suppose you have a table like the following:

Order_part		
order_num	part_num	quantity
PK		NN
FK	FK	
1	1	110
1	2	275
2	1	152

Part number and order number, each a foreign key in this relation, also form the composite primary key.

Suppose you were to go the *part_number* table and delete the row defined by the primary key value 1. The keys for the first and third rows in the *order_part* table are now parentless because there is no row in the *part_number* table with a primary key of 1 to support them. Such a situation exhibits a loss of referential integrity.

Now, suppose you had a mechanism to prevent this from happening? If you attempt to delete the row with a primary key value of 1 from the *part_number* table, the database management system does not allow you to do so. This is the way Vantage maintains referential integrity.

Besides data integrity and data consistency, referential integrity has the following benefits:

Benefit	Description
Increases development productivity	It is not necessary to code SQL statements to enforce referential integrity constraints because Vantage automatically enforces Referential Integrity by means of declarative RI constraints.

Benefit	Description
Requires fewer programs to be written	All update activities are programmed to ensure that established declarative referential integrity constraints are not violated because Vantage enforces Referential Integrity in all environments: no additional programs are required.

Referential Integrity Constraints

The combination of the foreign key, the parent key (more accurately, a *candidate* key), and the relationship between them defined by the referential integrity rule is called the referential integrity constraint.

This is a constraint defined on a table column set (using the CREATE TABLE or ALTER TABLE SQL statements) that represents a referential integrity link between two tables.

A referential integrity constraint is always defined for a foreign key column set in the child table in a relationship.

Note that you cannot use UDT, Period, Geospatial, BLOB, CLOB, or XML columns to define a referential integrity relationship or other database constraint.

See [Designing for Database Integrity](#) for additional information about how referential integrity is essential to maintaining the integrity of databases.

Vantage provides two other features related to referential integrity constraints: batch referential integrity constraints and referential constraints. The basic differences among the different referential constraint types are summarized in the following table.

Referential Constraint Type	CREATE TABLE Syntax	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
<ul style="list-style-type: none"> Referential Constraint Temporal Relationship Constraint For more information, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182.	REFERENCES WITH NO CHECK OPTION	No	None
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	All child table rows must match a parent table row
Referential integrity constraint	REFERENCES	Yes	Row

Referential constraints and temporal relationship constraints do not enforce the referential integrity of the database. Instead, they signal the Optimizer that certain referential relationships are in effect between tables, thus providing a means for producing better query plans without incurring the overhead of system enforcement of the specified RI constraints.

You should specify referential constraints and temporal relationship constraints only when you enforce the integrity of the referential relationship in some other way, or if the possibility of query errors and the potential for data corruption is not critical to your application. See [Semantic Constraint Specifications](#) and the information about the CREATE TABLE column definition clause in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information.

Batch referential integrity constraints are externally identical to regular referential integrity constraints. Because they enforce referential integrity in an all-or-none manner for an entire implicit transaction, they can be more high-performing than standard referential integrity constraints. In most circumstances, there is no semantic difference between the two, only an implementation difference. Batch referential integrity becomes important when a single statement inserts many rows into a table, like the massive inserts that can occur with INSERT ... SELECT requests.

Example: Referential Integrity Constraints

The following CREATE TABLE statement defines the following referential integrity constraints on table_A.

- A column-level constraint on the foreign key column, column_A1, and the parent table (table_B) key column, column_B1.
- A table-level constraint on the composite foreign key column set (column_A1, column_A2) and the parent table table_C.

```
CREATE TABLE table_A (
  column_A1 CHARACTER(10) REFERENCES table_B (column_B1),
  column_A2 INTEGER
PRIMARY KEY (column_A1),
FOREIGN KEY (column_A1, column_A2) REFERENCES table_C);
```

According to this definition, the single-column primary key column_A1 must reference the primary key column column_B1 of the parent table table_B. table_B must have a primary key, of which one column is column_B1, having the following definition.

```
CHARACTER(10) NOT NULL
```

The composite foreign key (column_A1, column_A2) must reference the primary key of the parent table table_C. table_C must have a two-column primary key, the first column of which has the following definition.

```
CHARACTER(10) NOT NULL
```

The second column of the table_C primary key must have the following definition.

```
INTEGER NOT NULL
```

Rules for Referential Integrity Constraints

Referential integrity constraints must obey the following set of rules:

- The parent key must exist when the referential integrity constraint is defined.
- The parent key columns must be either a unique primary index (UPI) or a unique secondary index (USI).
- The foreign and parent keys must have the same number of columns and their data types must match.
- Duplicate referential integrity constraints are not allowed.
- A foreign key value must be equal to its parent key value or it must be null.
- Self-reference (a condition in which the Parent and Child tables are the same table) is allowed, but the foreign and parent keys cannot consist of identical columns.

Domains and Referential Integrity

This topic describes various domain rules for primary (more accurately *candidate*) and foreign keys.

Definition of a Domain

A domain is a data type defined with explicit constraints. A data type is a well-defined set of values. For example, the data type INTEGER represents all the possible integer numbers (using a 4-byte two's complement representation), while an INTEGER column defined with an explicit CHECK constraint condition such as "count BETWEEN 25 AND 50" defines a domain on the set of numbers having the INTEGER data type. More unambiguous data types are usually application-specific and you can define them using the various facilities available to you for defining user-defined types and their associated constructs such as methods, orderings, and so on. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more details about user-defined data types.

Table Definition for Examples

An Employee table is used for the examples described in this topic. The definition of the Employee table is given by the following table.

Employee		
employee_number	last_name	supervisor_employee_number
PK,SA		FK
1	Smith	null
2	Brown	1
3	White	1
4	Gibson	2
5	Black	2
6	Jones	3
7	Mason	3

The following domain rule applies to the *employee_number* column: Its data type is INTEGER and any value inserted into the column must have a value greater than zero.

Domain Rules for Primary Keys

The following rules apply to primary key domains and referential integrity. The domain rules for primary keys apply equally, the necessary changes being made, to all candidate keys:

- New primary key values must be drawn from the domain set of all valid values for the column on which the primary key value is defined.
- You cannot define a primary key using any column defined with an XML, BLOB, or CLOB data type.
- You cannot delete a primary key that references foreign keys because that violates the referential integrity rule.

The rules for deleting a primary key that has foreign key references are as follows:

Rule Name	Rule Description
Prevent	<ul style="list-style-type: none"> • Do not delete a PK if it references an existing FK. • Do not change the value of PK if it references an existing FK.
Reassign	<ul style="list-style-type: none"> • Change the value of FK to a different PK value before deleting the old PK row. • Change the value of FK to a different PK value before changing the old PK value.
Nullify	<ul style="list-style-type: none"> • Change the FK value to NULL before deleting the old PK row. • Change the FK value to NULL before changing the old PK value.
Cascade	<ul style="list-style-type: none"> • Delete the FK row before deleting the old PK row. • Delete the FK row before changing the old PK value.

Each of these rules preserves the referential integrity of a PK-FK relationship.

Questions and Answers for Primary Key Domains

The following questions and answers about primary keys indicate some examples of how domain integrity is enforced. The questions refer to the employee table defined in the previous section, *Table Definition for Examples*.

Question	Answer	Explanation
Is it valid to add Employee 12345?	Yes	12345 is a positive integer.
Is it valid to delete the row for Employee 5?	Yes	Employee 5 is not referenced by any other row.
Is it valid to add Employee -23.67?	No	-23.67 is a negative decimal number.
Is it valid to delete the row for Employee 2?	Yes	Cascade rule.

Domain Rule for Foreign Keys

Foreign key values can be added only when they are drawn from the set of existing primary key values.

Questions and Answers for Foreign Key Domains

The following questions and answers about foreign keys indicate some examples of how domain integrity is enforced. The questions refer to the employee table defined in the previous section, *Table Definition for Examples*.

Question	Answer	Explanation
Is it valid to have Employee 6 report to Employee 4?	Yes	Employee 4 is drawn from the set of existing employee numbers.
Is it valid to delete the row for Employee 7?	No	Employee 7 has an assigned foreign key value.
Is it valid to have Employee 6 report to Employee 8?	No	Employee 8 is not drawn from the set of existing employee numbers.

Normalization and Database Design Problems

Although normalization is the only component of database design based in provably correct mathematics, it also retains some of the more subjective elements common to other aspects of database design. For example, it is usually possible to normalize a database schema in multiple ways. Normalization theory does not guarantee, or even suggest, that it produces a canonical set of relations for any given database. Instead, it guarantees that a normalized database schema is free of a number of problems that otherwise cannot be excluded, the most widely known of these problems being update anomalies.

This topic describes some of the problems commonly encountered in database design that cannot be solved merely by fully normalizing the database schema.

Preserving Functional Dependencies

It is often true that a relation can be nonloss-decomposed in more than one way; however, some of those ways are better than others. For example, consider the following relvar:

employee		
emp_num	dept_num	budget
PK		

This relvar has the following FDs:

- emp_no → dept_no
- dept_no → budget
- emp_no » budget

where » indicates the dependency is transitive.

A relvar like *employee* suffers from certain well known update anomalies that can be avoided by nonloss-decomposition into various projections. Relvar *employee* can be nonloss-decomposed into two valid 5NF projections.

For example, consider the following set of 5NF projections:

emp_dept	
emp_num	dept_num
PK	FK

dept_budget	
dept_num	budget
PK	

Call this decomposition *A*. The second valid 5NF decomposition of *employee* is as follows:

emp_dept	
emp_num	dept_num
PK	

emp_budget	
emp_num	budget
PK	

Call this decomposition *B*.

Note that the projection *emp_dept* is the same for both decompositions *A* and *B*.

Even though both decompositions are valid, decomposition *A* is better than decomposition *B*. For example, it is not possible to represent the fact that a given department has a budget unless that department also has at least one employee in decomposition *B*.

You can make the determination that decomposition *A* is better than decomposition *B* solely on an analysis of the FDs in the original *employee* relvar. Note that the projections in decomposition *A* correspond to the nontransitive FDs in *employee* (indicated by \rightarrow). As a result of this, you can update either projection without regard for the other as long as the referential constraint from *emp_dept* to *dept_budget* is satisfied. Provided only that the update in question is legal within the context of the given projection, which means only that it must not violate the primary key constraint for that projection, the join of the two projections after the update is still be a valid value for the *employee* relvar. In other words, the join cannot possibly violate the FD constraints on *employee*.

In decomposition B, by contrast, one of the two projections corresponds to the transitive dependency in *employee* (indicated by »). As a result, updates to either projection must at least be monitored to ensure that the FD *dept_no* → *budget* is not violated (in other words, if two employees have the same department, they must also have the same budget - consider, for example, the machinations required in decomposition B to move an employee from department D1 to department D2.)

The problem with decomposition B is that the FD *dept_no* → *budget* spans two relvars. On the other hand, while it is true that the transitive FD *dept_no* » *budget* in Decomposition A spans relvars, it is also true that the FD is enforced by default as long as the FDs *emp_no* → *dept_no* and *dept_no* → *budget*, which do not span relvars, are enforced, which only requires enforcement of the corresponding primary key uniqueness constraints.

The point being made is that you should decompose relvars in such a way that functional dependencies are preserved.

Full Normalization and Dependency Preservation

Sometimes the goals of decomposing to full normalization and decomposing them in a way that preserves functional dependencies can conflict. Consider the following example relvar:

student_subject_teacher		
student	subject	teacher
CK		
	CK	
Smith	Logic	Alonzo Church
Smith	Physics	Robert Millikan
Jones	Logic	Alfred Tarski
Jones	Physics	Max Planck

The predicate for the *student_subject_teacher* relvar is as follows: the student named *student* is taught the subject named *subject* by the teacher named *teacher*.

Assume the following restrictions for purposes of illustration:

- For each subject, each student of that subject is taught by only one teacher.
- Each teacher teaches only one subject, but each subject is taught by several teachers.

What are the FDs for the *student_subject_teacher* relvar?

- The first bullet implies the following FD:
student, subject — *teacher*
- The second bullet implies the following FD:
teacher — *subject*
- The second bullet also implies that the following FD is not true:

subject — teacher

For the sake of illustration, this set of dependencies is portrayed symbolically as follows: *student — teacher — subject*.

As the representation of relvar *student_subject_teacher* indicates, it has two candidate keys, the composite column sets *student-subject* and *student-teacher*.

Both CKs have the property that all columns of *student_subject_teacher* are functionally dependent on them, and that property no longer holds if any column is discarded from either column set.

Note, too, that *student_subject_teacher* also satisfies the FD *teacher — subject*, which is not implied by either CK. This absence of implication is indicative that the *student_subject_teacher* relvar is not in BCNF. As a result, *student_subject_teacher* suffers from certain update anomalies. Suppose, for example, that you want to delete the information that Jones is studying physics. You cannot do this without also losing the information that Max Planck teaches physics.

As usual, you can avoid the update anomalies by performing a nonloss decomposition into projections. In this particular case, the projections, both of which are in 5NF, are as follows:

student_teacher

student	teacher
PK	
Smith	Alonzo Church
Smith	Robert Millikan
Jones	Alfred Tarski
Jones	Max Planck

teacher_subject

teacher	subject
PK	
Alonzo Church	Logic
Alfred Tarski	Logic
Robert Millikan	Physics
Max Planck	Physics

It is now possible to delete the information that Jones is studying physics by deleting the row for Jones and Max Planck from projection *student_teacher* without losing the row for Max Planck and Physics from projection *teacher_subject*. So the problem is solved ... or is it?

Yes, that particular problem is solved. A remaining problem, however, is that this decomposition violates the FD preservation objective.

Specifically, the FD *student, subject* — *teacher* now spans two relvars, *student_teacher* and *teacher_subject*, so it cannot be deduced from the FD *teacher* — *subject*, which is the only nontrivial FD represented in *student_teacher* and *teacher_subject*. As a result, *student_teacher* and *teacher_subject* cannot be updated independently of one another.

For example, an attempt to insert a row for Smith and Max Planck into *student_teacher* must be rejected, because Max Planck teaches physics and Smith is already being taught physics by Robert Millikan, yet this fact cannot be detected without examining *teacher_subject*.

The point of this analysis is to highlight the fact that the following equally desirable objectives of database design can sometimes clash:

- Decomposing relvars to their ultimate normal form (or even just to BCNF)
- Decomposing relvars in such a way that preserve FDs

In other words, it is not always possible to achieve both objectives simultaneously.

Other points arise from this example:

- Neither design is superior to the other on its face. The principles of normalization suggest that one design is better, while the principle of FD preservation suggests that the other is.

As a result, the ultimate design choice must be based on other criteria.

- Assume that you settle on the two-relvar design.

In this case, the relvar-spanning FD must be declared, at least for documentation purposes, even if the design cannot enforce it.

A formal version of that declaration might look something like this:

```
FORALL s IN S, t1,t2 IN T, j1,j2 IN J
IF      EXISTS { S:s, T:t1 } IN ST
  AND EXISTS { S:s, T:t2 } IN ST
  AND EXISTS { T:t1, J:j1 } IN TJ
  AND EXISTS { T:t2, J:j2 } IN TJ
THEN j1 , j2
```

where the student, teacher, and subject domains are represented by S, T, and J, respectively.

If you think this example is overly contrived, and that involuted dependency structures such as *student* — *teacher* — *subject* never occur in practice, consider the following, more familiar, example:

address			
street	city	state	zip_code

This relvar satisfies the following set of FDs:

- *zip_code* — *city, state*
- *street, city, state* → *zip_code*

In other words, it is identical to the *student_subject_teacher* example, where *student* maps to the composite *street-city*, *state* maps to *subject*, and *zip_code* maps to *teacher*.

General Procedure for Achieving a Normalized Set of Relations

Although a designer having relatively little experience with normalizing a relational database can often put relations into 2NF and even 3NF without giving the process much thought, the following procedure is provided so that even a novice can pursue a set of steps through to completion and achieve normalization of a fairly simple set of relations.

The following procedure provides a high-level method for achieving a normalized set of relations:

1. Identify the attributes of the database.
2. Group related attributes into relations.
3. Identify the candidate keys for each relation.
4. Select the most useful primary key from among the set of candidate keys.
5. Identify and remove all repeating groups.

The result is a relation in 1NF.

6. If any of the resulting relations have identical primary keys, then combine them into a single relation.
7. Identify all functional dependencies between the attributes of a relation and its primary key.
8. Decompose the relations to a form where each nonkey attribute is dependent on all the attributes of the key. Do this by taking projections of the 1NF relation that eliminate any non-full functional dependencies.

The result is a set of relations in 2NF.

9. If any of the resulting relations have identical primary keys, then combine them into a single relation.
10. Identify any transitive dependencies in the relations decomposed to this point.
 - a. Examine relations for dependencies among nonkey attributes.
 - b. Examine relations for dependencies among key within the primary key.

11. Remove transitive dependencies by decomposition. Do this by taking projections of the 2NF relations produced by steps 8 and 9 that eliminate any transitive functional dependencies.

The result is a set of relations in 3NF.

12. If any of the resulting relations have identical primary keys, then combine them into a single relation if and only if no transitive dependencies result from the combination.
13. Remove any remaining functional dependencies from the 3NF set of relations. Do this by taking projections of the 3NF relations produced by steps 8 and 9 that eliminate any remaining functional dependencies where the determinant is not a candidate key.

The result is a set of relations in BCNF.

14. Remove any multivalued dependencies that are not also functional dependencies. Do this by taking projections of the BCNF relations produced by step 13.

The result is a set of relations in 4NF.

15. The likelihood that any join dependencies not implied by the candidate keys remain by this point is virtually nil. If you can identify any such relations, then take projections of them to eliminate the non-implied join dependencies.

The result is a set of relations in 5NF.

16. Particularly for temporal data, but sometimes to eliminate nulls as well, take projections to 6NF.

Advantages of Normalization for Physical Database Implementation

The following list summarizes the advantages of physically implementing a normalized logical model:

- Greater number of relations
 - More primary index choices
 - Optimal distribution of data
 - Fewer full-table scans

For example, consider the multicolumn primary index of a Fact table, which has the primary key of each of its dimension tables as a component. The Optimizer cannot retrieve a row with a partial primary index, so many, if not all, accesses to the Fact table must use a full-table scan.

This assumes that you implement the natural primary key of the fact table as the primary index. If you instead define a surrogate key column to be the primary index, the full-table scan issue is moot (see the definition for *Surrogate key* in [Definitions](#)). This practice is not generally advised.

- More joins possible
- Enhanced likelihood the Optimizer will use the high-performing merge or nested join methods
- Optimal data separation to eliminate redundancy from the database
- Optimal control of data by eliminating update anomalies
- Fewer columns per row
 - Optimal application separation
 - Optimal control of data
- Smaller rows
 - Optimal data blocking by the file system
 - Reduced transient and permanent journal space
 - Reduced physical I/O

How Normalization Is Beneficial for Physical Databases

The fundamental objective for a relational database management system is to keep data independent of the applications or analysis that use it. There are two reasons for this:

- Different applications require different views of the same data.

- Data must be extendable, and a framework must exist to support the introduction of new applications and analyses without having to modify existing applications.

This topic develops some of themes described elsewhere in this document by explaining their relevance to everyday exploratory data analysis problems in data warehousing.

Data Dependence and Data Independence

The main objective of relational DBMSs is data independence. For years, the relational database management systems used to run businesses, often referred to as OLTP systems, made data independence obligatory. In an OLTP database, data is stored in nonredundant tables that demand that every column of the table be rigorously related to its primary key alone and to no other tables. This ensures that information is available to all applications and analyses that use it, and it provides a mechanism for maintaining consistency and reliability across applications: a single source of each particular data element, a single version of the truth.

Data independence works well for OLTP systems because the applications accessing the data generally access single tables or join only a few, small tables in relatively simple queries. With the introduction of the data warehouse, previously unheard of demands were placed on the relational database management systems underlying them. In the data warehouse environment, large tables must be scanned and large result sets are frequently returned. Many tables are joined together, complicated calculations are made, and detailed data is aggregated directly in the queries. In addition, large data volumes are extracted, transformed and loaded into the tables concurrently with users running queries against the data. It quickly became apparent that databases created and tuned for OLTP could not sustain the performance levels required to support the demands of business intelligence processing. The OLTP databases could not perform the queries within their allotted time window or, in some cases, at all.

This situation highlights the potential for contradiction between designing databases for optimum integrity and designing databases for optimum performance. The key to data independence is data normalization, and normalized data schemas are the most demanding of system performance.

To address the issue of poor performance, data independence has often been abandoned in many environments and denormalized schemas have been used to address a few particular, rather than all general, analytical needs of the enterprise.

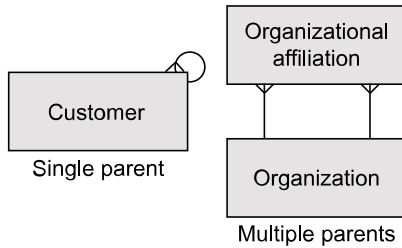
Although this arrangement addresses short-term decision support needs, it compromises the enterprise view of the data and its adaptability. Data independence, adaptability, and cross-enterprise functionality go hand in hand, and a normalized data schema is critical to reaching these objectives.

The following topics provide some detail about why this is true.

Recursive Relationships

The star schema (see [Dimensional Modeling, Star, and Snowflake Schemas](#)), which is the most common form of denormalization used in contemporary data warehousing, cannot handle every kind of relationship that can exist comfortably in a fully-normalized environment. Recursive relationships are one such example. Recursion, as the term is generally used in computer science, is only a small subset of the recursive function theory of formal logic.

A recursive relationship exists when the parent of a member in a hierarchical relationship is also a member of the same entity. As demonstrated by the following figure, there are two ways that this can manifest itself: with only a single parent or with multiple parents:



The most commonly used example of a single-parent recursive relationship is an *employee* table, where both an employee and its manager have rows. From an E-R perspective, you would say a manager has employees. But managers, too, are employees. This also means that managers can have managers who are employees, and so on.

In the diagram, the single-parent recursive relationship is a *customer* table in which a customer can be a customer of yet another customer in the table. The classic multiple-parent recursive relationship is the bill of material. The diagram shows an example in which multiple organizations can have multiple organizational affiliations. Project work breakdown hierarchies are another common example of a multiple parent recursive structure.

In a recursive structure, there can be an unlimited number of levels without knowing how many levels each member hierarchy currently has or potentially can have. One hierarchy have only two levels, while another might be 15 levels deep. Herein lies the limitation of the star schema for handling recursive relationships: it requires a fixed number of levels, because each level is set up by a series of fixed columns in a dimension table. Because you do not know the number of levels in a recursive structure, you cannot predefine the columns.

The most critical entities in an enterprise data model frequently have recursive structures. Organizational hierarchies such as internal, customer, supplier, and competitor entities are usually recursive relationships.

Arguments Against Denormalizing the Physical Database Schema to Increase Its Usability

Many data warehouse designers argue that denormalized physical database schemas are easier for end users to navigate than fully normalized schemas.

Denormalized physical schemas certainly seem more user-friendly than the complexity of a highly generalized, normalized data model. However, denormalized physical schemas are driven by known queries, so their ease of use is somewhat illusory. Formulating queries to address novel requirements, a task that is nearly definitive of the data warehouse process model, is made more difficult, if not impossible, in a denormalized environment. A fully normalized enterprise data model is flexible enough to support the undertaking of any new analyses of the data.

That said, the reality is that end users typically do not write queries anyway, and when they do, they are likely to use a third party natural language query generator, so the usability argument is often moot. Coding novel queries is often the responsibility of an application developer or a natural language query writing tool.

More importantly, you can create “denormalized” views to implement a semantic layer that makes the normalized data model easier to navigate (see [Denormalizing Through Views](#)). Few sites permit users to query base tables directly anyway, so creating views on base tables that look exactly like their denormalized table counterparts should not be an issue.

If there were no issues of performance for those database management systems that lack the parallel processing power of Vantage, then denormalization could be handled universally by implementing views. Star schemas, snowflakes, summary tables, derived data, and the like could be built as virtual clusters of tables that look exactly like their physical counterparts. By handling denormalization virtually, the relationships within, between, and among the underlying base tables of the schema remain intact, and referential integrity can be maintained by the system regardless of how many virtual denormalized relationships are created. This flexibility frees DBAs to create any number of denormalized views for users while simultaneously maintaining semantic data integrity and eliminating the data redundancies required by denormalized physical schemas.

DBAs can create virtual, subject-oriented schemas for specific applications as well as creating views for more general database access without affecting the underlying base table data. These same views can also be used to enforce security constraints for the different communities of business users who must access the database.

Consider another argument that favors the ease of use of a normalized database schema over a denormalized schema. A physical star schema has physical dimensions that support a physical fact table. However, for some dimensions there can be mutually exclusive substitutes for the same data. For example, suppose an airline is interested in both the point-to-point travel of customers between segments in addition to their travel between their true origins and destinations. This discussion abbreviates this dimensional family as O&D.

The true O&D dimension is different from the segment O&D, although superficially, it looks the same. Moreover, their respective consolidation of facts is different as well, although the detailed base table data is the same. If the star schemas are physicalized, two very large schemas must be created, maintained, and coordinated to represent them, whereas with virtual star schemas, the data is maintained only in the base tables, producing a single, consistent version of the truth.

Arguments Against Denormalizing for Performance

Data warehousing authorities often argue that physical denormalization of the logical data model offers better performance than a physical instantiation of the normalized logical data model. This question is mitigated when the SQL Engine can handle a normalized physical design and scale linearly. Vantage does just that.

If further need to improve performance remains aside from scaling the database, then implement a well-planned data propagation strategy that maintains and complements the underlying normalized base table substructure.

To begin, consider propagating denormalized data within the same database instance. Weigh propagating to another environment only if there are other considerations for the target data source, such as geographic needs or the need to support proprietary data structures. In any case, the propagation is from the data warehouse and not directly from source systems.

These reasons support this strategy.

- The fully-parallel capabilities of the data warehouse can be used to optimize the propagation of data.
- By keeping the data in the same instance of the database, you can perform hybrid queries that take advantage of both the denormalized and normalized data. For example, large volumes of nonvolatile data from transaction detail rows can be propagated into new physical fact tables, and smaller volume, highly volatile dimensional data can be built into virtual dimension tables.
- The resulting administration of the complete data warehousing environment is not only easier, but also less expensive.

Denormalized Physical Schemas and Ambiguity

A denormalized physical schema creates ambiguity because within a denormalized table, it is not possible to determine which columns are related to the key, to parent tables, or to one another (see [Functional, Transitive, and Multivalued Compatibilities](#), [The Referential Integrity Rule](#), and [Domains and Referential Integrity](#) for explanations of how fully normalized tables avoid this ambiguity). Normalized models maintain all relationships through the association of primary keys with their functionally dependent attributes and with foreign keys.

Consider the following non-normalized table, where all but one of the column names have been abbreviated to fit on the page.

com_id	com_name	grp_id	grp_name	fam_id	fam_name	sequence
PK						

The expanded column heading names are given in the following table.

Abbreviated Column Name	Actual Column Name
com_id	commodity_id
com_name	commodity_name
grp_id	group_id
grp_name	group_name
fam_id	family_id
fam_name	family_name

This table meets the criteria for 2NF, but not for 3NF, because not only does every non-key attribute not depend on the primary key, but several non-key attributes are functionally dependent on other non-key attributes (see [Third and Boyce-Codd Normal Forms](#)).

You cannot perceive the relationships among the *commodity_id*, *group_id*, and *family_id* attributes by looking at this table. While it is fairly obvious that *commodity_name* is a functional dependency of the primary key, it could also be true that the commodity attribute has separate relationships with groups and families, or it could be true that commodities are related to groups, which are, in turn, related to families. Moreover, the relationship of the Sequence attribute to the primary key, or to anything else, cannot be resolved. Is it used to order commodities within groups, commodities within families, or groups within families? It is surprising how often such dangling relationships are found in denormalized logical models.

In contrast, normalized models maintain all relationships through the association of primary keys with their attributes and with foreign keys. As a result, you can clearly see the relationships among the various tables in the database schema simply by looking at the data model.

Referential Integrity and a Denormalized Schema

Referential integrity cannot be effectively maintained within a denormalized database schema.

By definition, denormalized structures compromise data quality. Relational database management systems enforce referential integrity to ensure that whenever foreign keys exist, the instances of the objects to which they refer also exist. Denormalized tables cannot support referential integrity through declarative constraints. For example, users have no guarantee that the children in a denormalized table have parents in that table unless they implement some other, more costly method of programmatically ensuring semantic data integrity. The problems with maintaining referential integrity by means of application code rather than through declarative constraints are myriad (see [Designing for Database Integrity](#)).

Also, as new incremental data loads are inserted into the database, changed fields are not reflected in the old data. When an attribute is changed in the source system, only the newly loaded rows reflect those changes. For example, suppose the marital status, last name, or another field in the customer table changes. Any new loads for that customer reflect the different data in the changed fields, causing inconsistencies and incorrect results in certain types of analyses because both the changed and unchanged attributes are stored redundantly, violating the concept of a single version of truth that is so easily maintained with a fully normalized database schema.

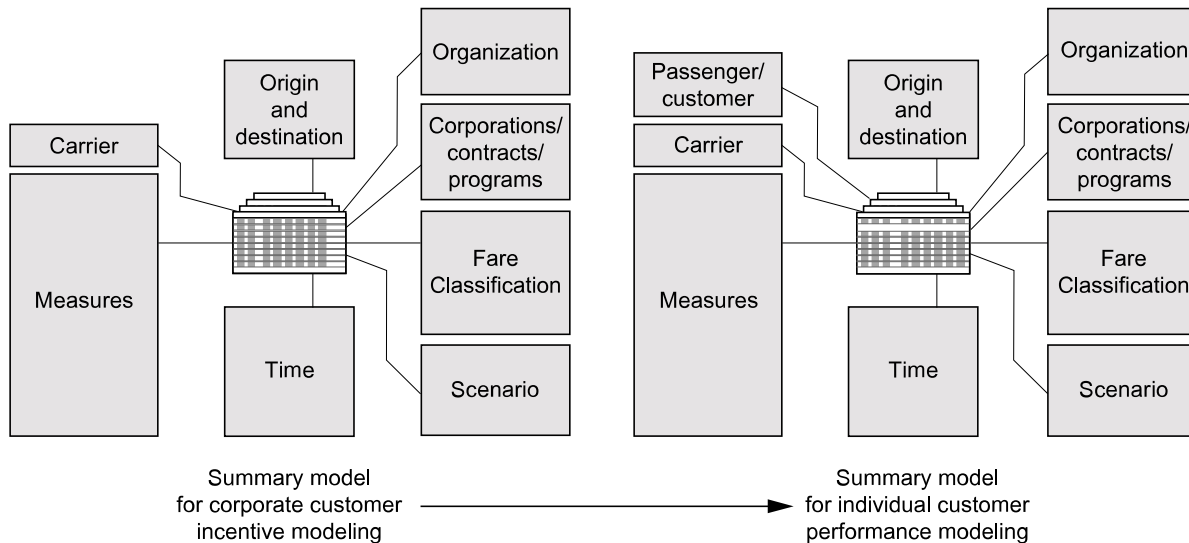
Denormalized Views Versus Physical Denormalization of the Database Schema

A fully-normalized database schema accessed through denormalized views is far more flexible in creating and managing dimensional consolidation hierarchies for OLAP models than a star schema.

The dimensional view approach, by using one physical database, eliminates both the additional costs and the data quality problems of maintaining redundant data.

DBAs who deal with denormalized methods typically define a single set of mutually exclusive dimensions for a given OLAP model. However, it is possible to create many nonexclusive dimensional paths for an individual OLAP model that also are reusable across many different OLAP models.

The following figure shows two denormalized virtual schemas that share a number of common dimensional views:



The number of virtual schemas that can be created and aligned with particular users or user communities is unlimited. As DBAs become more sophisticated in their application of this concept, they can manage an extensive number of database views tailored toward the specific needs of individual users as well as the specific needs of various user groups. It is not possible to maintain such an extensive schema management program when the data must be physically propagated for performance reasons.

Consider another example: suppose there are eight potential dimensional consolidation paths that could arise from the following three entities:

- Households
- Customers
- Accounts

In OLAP terminology, a *consolidation path* is a set of hierarchically related data. Consolidation itself is the process of aggregating the hierarchical data to form subtotals. The highest level in the consolidation path is the dimension for the data.

The eight possible consolidation paths for this data are the following, where → indicates a downward path through a hierarchy and in E-R terminology is equivalent to the word *have*:

- Households
- Customers
- Accounts
- Households → Customers → Accounts
- Households → Accounts → Customers
- Households → Customers
- Households → Accounts

- Customers → Accounts

When either these consolidation paths or the previously mentioned schemas are created virtually, any problems with managing referential integrity are rendered moot because the only data that must be maintained is in the base tables. However, when these tables are instead instantiated as separate physical tables, managing referential integrity across the multiple alternate dimensional paths becomes very difficult to maintain. As more dimensional entities and potentially hundreds more dimensional consolidation paths are added to the schema, it becomes impossible to continue to propagate data physically. As a result, if a physical star schema is implemented, compromises to the maintenance of the semantic integrity of the data must be made to accommodate the resulting complexity.

Dimensional Analysis

A star schema is designed specifically to support dimensional analysis, but not all analysis is dimensional. Many types of quantitative analysis, such as data mining, statistical analysis, and case-based reasoning, are actually inhibited by a physical star schema design.

Given the definition of 3NF as the key, the whole key, and nothing but the key, any attributes that do not modify the key, the whole key, and nothing but the key are “noise” in the context of normalization theory.

If you consider the household/customer/account example of the previous section (see *Denormalized Views Versus Physical Denormalization of the Database Schema*) in a different light, you encounter an interesting problem with denormalization. An OLAP modeler, focused on creating dimensional views and drill-downs, is likely to assume that a household can have many customers but that a customer can belong to only one household. However, the underlying normalized data model might reveal a many-to-many relationship between households and customers. For example, a customer can belong to many different households over time.

A modeler focused on OLAP applications typically is not concerned with this relationship because it is unlikely that OLAP users would want to drill down into previous households. However, a data mining user might find tremendous value in exploring this relationship for patterns of behavior that revolve around the changing composition of households.

For example, consider the following possible household change behaviors:

- Children leave their parents to start their own households.
- Couples marry and form one household from two.
- Married couples divorce and form two households from one.

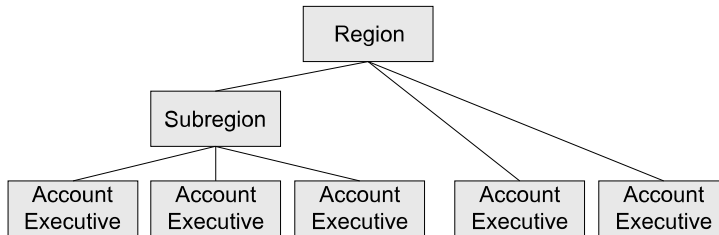
And so on.

The enterprise needs a logical model to address the precise needs of any analytical methodology, whether it be a complicated ad hoc SQL query, an OLAP analysis, exploratory work using data mining, or something entirely different.

Unbalanced Hierarchies and Unnormalized Physical Schemas

You cannot make the universal assumption that all levels in a dimensional model are balanced. A star schema dimension table, which requires fixed columns for each level, has considerable difficulty in handling unbalanced hierarchies.

An unbalanced hierarchy, as shown in the following figure, is a hierarchy in which a particular level can have its parent at different levels above it. An account executive, for example, might report to a region or a subregion.



Because a star schema creates specific columns to handle each level in a hierarchy, it requires that all hierarchies be balanced so that, using the current example, an account executive would always need to be at level three.

Analyses Driven By Normalized Relationships

Many types of analysis are driven by the normalized relationships in the database.

There is, for example, a considerable amount of analysis that is not quantitative in any way. For example, market basket studies determine clusters of products that are most frequently purchased together. A market basket study does no counting, aggregating, or any other quantitative measure or analysis.

Tracking the cycle times for various entities requires a sophisticated method of following the relationships between statuses and events. Modeling strategic frameworks such as value chains, supply chain management, competency models, industry structure analysis, and total quality management requires sophisticated relational models for which denormalized approaches are inadequate.

Costs of Denormalization

In the long term, the actual cost of maintaining a denormalized environment exceeds the costs of a normalized environment, and the cost of decreased cross-functional information opportunities is much greater.

There are economic costs associated with denormalization that often are not considered. In a star schema, each row in a dimension table contains all of the attributes of every entity mapped within it. In the accounts, customers, and households example described in the previous section, you would carry all the redundant household data and all the redundant customer data for each account. Not only does the redundancy from the expansion of columns exist, but in the case of the many-to-many relationship between accounts and customers, the number of rows also increases because a separate row is required for each legitimate account-customer combination. When there are millions of accounts, thousands of which are jointly held, this horizontal and vertical redundancy can add significant storage overhead.

These costs generally are not significant in light of performance gains, but they are significant in terms of the additional DBA and application coding costs incurred to programmatically maintain referential integrity.

More important is the cost of business opportunities lost because of compromises that render entire categories of data analysis difficult, if not impossible, to perform. Total benefits are much greater for the

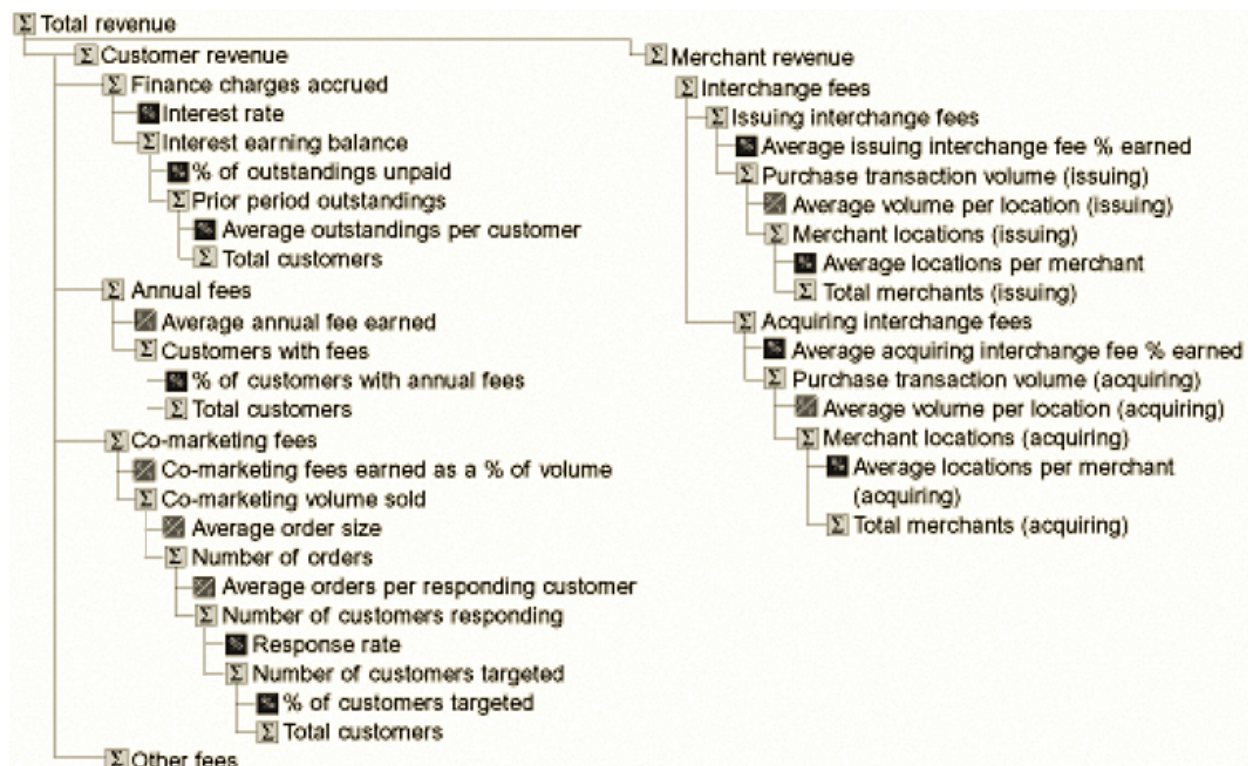
normalized approach because of its adaptability and generalizability. As an enterprise begins to define its business analysis requirements, it initially identifies only a fraction of what it ultimately needs. Possibly as much as 95% of the real analytic needs of the enterprise go undefined in the infancy of the data warehouse.

The key to being able to capitalize on unsuspected opportunity is flexibility. By building an adaptable database schema from the beginning, an enterprise enables itself to address new business needs as they are identified without having to compromise or restructure the database. The enterprise is also able to address any new challenges in the shortest time frame because it does not need to involve its IT staff in designing, building, and propagating data for the new queries. The faster a company can respond to unexpected challenges and opportunities, the higher the business value that can be realized from its data.

Cross-Functional Analysis and Denormalization

Cross-functional analysis becomes increasingly difficult as a database becomes more and more denormalized.

Data warehousing provides a means to build complex interrelated models for cross-subject area analyses in ways no other system can. These models can move beyond the traditional financial measures to begin interrelating internal process measures and customer-oriented measures as well. More importantly, these more sophisticated analytical models can begin to push from results-oriented or outcome-oriented measures toward measures directly linked to organizational activities. The following figure shows an example of interrelated measures from a credit-card model.



The problem with denormalized data for this example is that measures cross many different denormalized schemas. Even though Vantage is optimized to handle them, star joins are cumbersome to process, and

making joins across multiple models also makes them extremely complex. In this particular example, there could easily be four or more distinct star schema structures.

No Crystal Ball

There is no crystal ball for predicting the future data analysis needs of the enterprise. The tendency to compromise and build the enterprise data model based on current needs is much greater for those building denormalized enterprise database schemas than for those building normalized schemas.

Even when aware of the compromises being made, individuals undertaking a denormalized implementation of the logical model tend to discount the possibilities of those compromises working against future growth of the system because they believe they have already anticipated all future analytical opportunities.

However, as business discovery evolves, it inevitably becomes clear that those initial projections do not fully support the requirements and vision of the business.

Design For Decision Support and Tactical Analysis

Not only is basic schema normalization an important design consideration, but normalization tailored specifically for the analytical needs of business intelligence is also critical. Even normalized data warehouses are often built as if they were designed to support OLTP applications.

Designers often fail to examine the following list of considerations:

- Integration of subject areas
- Versioning across time
- Generalization of key entities in anticipation of change
- Interrelation of life cycles across many cross-subject area entities and events
- Integration of measures, calculations, and dimensional context paths across the enterprise

These factors all support the immediate and long-term benefits provided by a fully normalized enterprise data model. The fully normalized schema provides a framework both for continued growth and for increasing user insight.

The moral of the story is this: it is often a mistake to compromise your logical data model when you select tools to extend the value of your data and to optimize the delivery of information to your users. When necessary to facilitate ease of use or to improve tool performance, implement views to help ensure the continued integrity of your logical data model. When you are evaluating the data model for your data warehouse, ensure that model not only supports your business today, but that it is capable of supporting the enterprise well into the future.

Design for Flexible Access Using Views

You should design your applications to access the database through views rather than directly accessing base tables. Administrators sometimes avoid application designs that access the database through views because of a perceived performance penalty. In practice, this is rarely true for access operations, and then only in exceptionally rare cases. In fact, view projections of their underlying base tables can minimize the use of spool space by joins by reducing the number of columns that must be redistributed to make the join.

Furthermore, you can more readily control locking for access through careful view design, and so minimize the complexity of applications that access the database through those views.

As a general rule, the best practice is to design applications to access the database only through views rather than accessing base tables directly.

Besides providing you with the ability to provide pseudo-denormalized access for enhanced usability (see [Denormalized Views Versus Physical Denormalization of the Database Schema](#)), views also provide a degree of data independence that permits you to make changes either to your applications or to the physical database those applications access without having to worry about rewriting application software. Views can also provide a coarse level of schema versioning that is otherwise not available.

The Activity Transaction Modeling Process

This section first describes and defines some of the concepts underlying the Activities and Transaction Modeling (ATM) process and then describes the ATM process and its forms, providing examples of how to fill them out and apply them to the physical database model.

You use the early ATM process forms as input to more complex forms later in the process. You use the final ATM process forms when you create the physical objects in your database, ensuring that the universe of domains, constraints, table accesses, and reports and queries are understood well in advance of the process of creating the physical database objects that derive from those constructs.

The ATM process is the first step in the transition from your logical data model to the implementation of your physical data.

The second and final step in that transition is to collect or prototype data demographics.

Goals of the ATM Process

1. Define all domains and constraints.
2. Identify all applications.
3. Model application processing activities including their transactions and run frequencies.
4. Model each transaction using the following information.
 - a. Identify tables used.
 - b. Identify columns required for value and join access.
 - c. Estimate qualifying cardinalities.
5. Summarize value and join access information across all transactions.
6. Add data demographics to the Table Access Summary by Columns report.
 - Table cardinalities
 - Column value distributions (histograms)
 - Column change ratings

The following table indicates the activities of this process and the forms required to complete each activity.

Step	Action	Form Used
1	Define all domains in the system.	Domains
2	Define all constraints for the system.	Constraints
3	Identify all applications in the system.	System
4	Model each identified application.	Application
5	Model each identified transaction.	Report/Query Analysis

Step	Action	Form Used
6	Summarize all value and join accesses.	Table
7	Transfer access information.	
8	Compile or estimate data demographic information.	
9	Identify column change ratings.	

Note that row sizing calculations are covered in [Row Size Calculation](#).

Terminology Used in the ATM Process

Term	Definition
<i>Domain</i>	<p>A well-defined, closed set of values from which column data can be drawn. Domain is really just another term for data type.</p> <p>Predefined data types generally do not provide sufficiently distinct domains, and you should consider using distinct user-defined types whenever domain integrity is important to your databases or applications. Note that you cannot specify any type of constraint for a UDT column. You can otherwise use various constraints, particularly check constraints, to restrict the range of values a column will accept, so if your application requires any kind of check, uniqueness, or referential constraints, you cannot define a domain for the column using a distinct type.</p> <p>See <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 and <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147 for information about creating UDTs and their associated database objects.</p> <p>Note that Vantage does not support the physical concept of domains.</p>
<i>Constraint</i>	<p>A well-defined physical restriction that can be defined for a column or table.</p> <p>Constraints include the following:</p> <ul style="list-style-type: none"> • UNIQUE • PRIMARY KEY • FOREIGN KEY • CHECK <i>expression</i> • REFERENCES <p>The following rules and recommended practices apply to constraints.</p> <ul style="list-style-type: none"> • Always name constraints. • Constraint names must be 30 or fewer characters. • A constraint name must be unique among all other constraint names defined for a table. • You can specify constraints both at the column and table levels. • The system does not assign names to constraints you do not name. • You cannot define any type of constraint on an XML, BLOB, or CLOB column. <p>For more information about the various constraints, see CREATE TABLE information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i>, B035-1184.</p> <p>For design-related information about column and table constraints, see Designing for Database Integrity.</p>

Term	Definition
<i>Table</i>	<p>A multidimensional, abstract representation of an entity constructed from the following components:</p> <ul style="list-style-type: none"> • Rows, representing tuples • Columns, representing attributes <p>Tables are sometimes referred to as relations, though the correspondence between relations and tables is not always direct.</p> <p>Example: The Location Entity Before It Is Fully Attributed shows the structure of the Location entity before it is fully attributed.</p>
<i>Row</i>	<p>An instance of an object in a relational table. Rows in relational tables are not ordered. The expression is a synonym for record, a term not used for relational database systems. Rows are sometimes referred to as tuples, though the formal term for the corresponding concept is <i>n</i>-tuple.</p> <p>The number of rows in a table is referred to as its cardinality.</p> <p>Example: A Randomly Selected Row From the Location Entity indicates a randomly selected row from the <i>Location</i> entity.</p>
<i>Column</i>	<p>A unique, atomic attribute of a relational entity. Columns in relational tables are not ordered logically, though they are ordered physically and can be referred to by their column number. Columns are sometimes referred to as attributes.</p> <p>The number of columns in a table is referred to as its degree or arity.</p> <p>Example: CustNum Column From the Location Entity indicates the <i>CustNum</i> column from the <i>Location</i> entity.</p>
<i>Primary Key</i>	<p>The primary key is a column set that uniquely identifies a tuple within a relation. Every relation must have one and only one primary key defined during logical design, but a primary key need not be formally defined for a table corresponding to a logically defined relation.</p> <p>A table can have multiple candidate primary keys, but only one defined primary key. A candidate primary key not selected as the primary key for a table is referred to as an alternate key.</p> <p>A primary key cannot be null and must be unique.</p> <p>No component of a primary key can have the XML, BLOB, or CLOB data types.</p> <p>Example: The Primary Key Column for the Location Entity indicates the primary key column for the <i>Location</i> entity.</p>
<i>Identity Column</i>	<p>A column for which the values are unique and system-generated. Values for identity columns can be generated by the system in all cases or only in those cases for which users do not provide a value.</p> <p>Identity columns are frequently used to generate surrogate keys.</p> <p>You cannot define an identity column having the XML, BLOB, or CLOB data types.</p> <p>For more information about identity columns, see System-Derived and System-Generated Column Data Types and the detailed description of identity columns and their use in CREATE TABLE information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i>, B035-1184.</p>
<i>Foreign Key</i>	<p>The foreign key is a column set that identifies a relationship between the table for which it is a foreign key and one or more other tables in the database.</p> <p>Foreign keys are used both as join conditions and to maintain referential integrity between tables.</p>

Term	Definition
	<p>A foreign key must be the primary key for the table it references and it can be null unless you define the foreign key column set for the table to exclude nulls.</p> <p>No component of a foreign key can have the XML, BLOB, or CLOB data types.</p> <p>Example: The Three Foreign Keys For the Location Entity indicates the three foreign key for the Location entity. Notice that the <i>CustNum</i> column, which is constrained to be not null, relates <i>Location</i> to <i>Customer</i>, the <i>State</i> column relates <i>Location</i> to <i>State</i>, and the <i>Country</i> column, which is also constrained to be not null, relates <i>Location</i> table to <i>Country</i>.</p>
<i>Normalization</i>	<p>A method for segregating the attributes of a database into individual tables in such a way that those attributes uniquely modify (or depend upon) the primary key for that table.</p> <p>Somewhat more formally, a relation is said to be fully normalized if all its nonkey attributes are functionally dependent on its primary key.</p>

Example: The Location Entity Before It Is Fully Attributed

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

Example: A Randomly Selected Row From the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

Example: CustNum Column From the Location Entity

LocationNum	CustNum	State	Country
-------------	---------	-------	---------

PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

Example: The Primary Key Column for the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

Example: The Three Foreign Keys For the Location Entity

LocationNum	CustNum	State	Country
PK, SA	FK, NN	FK	FK, NN
2	10	CA	USA
7	0	CA	USA
1	2	NY	USA
4	7	PR	USA

Domains

The concept of domain as used in relational database theory derives directly from the more formal definition given in set theory and function theory. A variable is defined as a set of points having both a domain and a range, where the domain defines the specific type of data represented by the set and the range defines the bounds on that type.

A more concise way of expressing this is to say that domains are data types. It is frequently a good idea to extend this definition to include constraints defined for a column. Extended domains of this type are frequently referred to as business rules. A simple example would be to define employee_number as an INTEGER type and extend the definition with a constraint to disallow negative integer numbers as valid

employee numbers. Such an example conflates the mathematical notions of range and domain. Strictly speaking, a domain is only a data type, and range constraints are not part of its definition.

Domains have been a fundamental concept supporting the theory of relational databases since the inception of that theory. The variables described by a domain in a relational database are the valid values an attribute, or column, can have.

Unfortunately, the ANSI/ISO SQL standard does not support a rigorous atomic definition of domain, so it is up to the database designer to define the domains for a database and their ranges rigorously by creating various constraints on table columns, by creating appropriate user-defined data types (the behaviors of distinct UDTs are based on the behaviors of the predefined data types from which they are derived. Distinct UDTs can optionally also have user-defined behaviors.

The behaviors of the other category of user-defined data types, structured UDTs, are exclusively user-defined. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about creating distinct and structured UDTs and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details about user-defined data types and their associated database objects), and by ensuring that applications do not make nonsensical cross-domain data manipulations such as subtracting an integer part number from an integer inventory count (see [Column Comparisons](#)).

Note:

You cannot specify a referential integrity constraint for a UDT or CHECK constraint column, though UDT columns (with the exception of ARRAY, VARRAY, Period, XML, JSON, and Geospatial types) can be used in the definition of a UNIQUE or PRIMARY KEY constraint or a primary or secondary index. You can otherwise use various constraints, particularly CHECK constraints, to restrict the range of values a column will accept, so if your application requires any kind of check, uniqueness, or referential constraints, you cannot define a domain for the column using a distinct type.

Domain Properties

Domains in the SQL Engine are defined to have the following properties:

- A domain defines the set of possible values that can be written to a table column.
- The values of a domain cannot be decomposed to component values; they are atomic.

Note:

This is not true for domains having INTERVAL, TIMESTAMP, or structured UDT data types, which are not atomic, but the concept generally holds true otherwise.

- A domain must have a name.
- A domain must have an assigned data type.

Domains and Keys

The following rules apply to these relationships.

- All primary key values for a table are drawn from the domain that defines all possible values for that column.

That domain cannot be defined with an XML, BLOB, or CLOB data type.

- All foreign key values that reference a primary key value in another table are drawn from the same domain as the values for that primary key.

That domain cannot be defined with an XML, BLOB, or CLOB data type.

Teradata Data Types

When your applications require a strict domain type to ensure domain integrity, particularly for column comparisons and arithmetic operations, you should consider defining user-defined distinct data types for those domains.

See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about creating user-defined data types and their associated database objects.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about writing external code for the methods you define to work with user-defined types.

The following table lists the valid Teradata predefined data types you can use to define a domain or to create new user-defined data types.

Data Type	Definition
ARRAY VARRAY	Not valid. You cannot create a distinct UDT using a ARRAY/VARRAY data types, so you cannot create a domain using an ARRAY/VARRAY predefined type as its base.
BLOB BYTE VARBYTE	A binary integer used to store digital images.
BINARY LARGE OBJECT (BLOB)	A large binary string used to store binary objects such as musical recordings, videos, and other multimedia.
CLOB CHARACTER VARCHAR LONG VARCHAR GRAPHIC	Any glyph from a supported language. For the English language, this type is often referred to as alphanumeric. The CLOB type is typically used to define large character objects whose length exceeds 64KB.
CHARACTER LARGE OBJECT (CLOB)	A large character string used to store documents, possibly encoded using tag languages such as XML.
XML	A large binary string used to store XML documents in XML format.
DATE	A valid, named 24-hour epoch from the Gregorian calendar.
DECIMAL	Any base-10 real number, including those with a fractional part.
NUMBER (Exact form)	Any base-10 real number, including those with a fractional part.

Data Type	Definition
FLOAT REAL DOUBLE PRECISION	A rational number expressed in exponential format which, depending on the value, might be exact or might be an approximation to a real number.
NUMBER (Approximate form)	Any base-10 rational number expressed in exponential format.
BIGINT INTEGER BYTEINT SMALLINT	Any natural number.
INTERVAL	A time duration with optional fractional precision. Interval data is not implemented atomically, though it is generally treated logically as if it were atomic.
TIME	A valid time expressed using 24-hour notation with optional fractional precision. The TIME type can also be defined with a TIME ZONE.
TIMESTAMP	A valid date and time expressed using 24-hour notation with optional fractional precision. The TIMESTAMP type can also be defined with a TIME ZONE.
Period	Not valid. You cannot create a distinct UDT using the Period data type, so you cannot create a domain using the Period predefined type as its base.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for complete listings and descriptions of the data types available for table columns.

You can also use these predefined data types to create your own user-defined data types that may be more suitable for your particular application workloads. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details.

Domains in the Logical Data Model

You should always define and assign domains to your table attributes during the logical design phase of your database design.

Domains can be bounded specifically, nonspecifically, or both simultaneously. For example, you can define the employee number domain to be both not null (a nonspecific boundary) and constrained within a restricted range such as greater than 100 and less than 1,000,000. Unfortunately, you cannot specify CHECK or any other types of constraints on UDT columns, which lessens their usefulness for defining domains.

Column Comparisons

The necessity for naming your domains becomes clear when you examine the topic of column comparisons.

According to the relational model, columns can be compared if and only if their values are drawn from the same domain. You cannot compare XML, BLOB, or CLOB columns using the built-in SQL operator set. You can write UDFs to make such comparisons, however, and if you create UDTs based on BLOB or CLOB types, you can create methods for those UDTs that would permit you can make such comparisons (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for further information). If an application is to compare data from different columns in any way, all of the following statements must be true for all of the columns being compared.

- They must share the same set of values.
- The same value represents the same real world object in all cases.
- The values can be compared, added, subtracted, and joined.

Note that you cannot operate on XML, BLOB, or CLOB data types

In terms of domains, these rules can be stated as follows.

- Columns from the same domain always have the same defined domain name, which can be based on a user-defined data type.
- Columns from the same domain always have the same defined constraints.
- Columns from the same domain always have the same data type.

Particularly in the case of user-defined data types, the domain and the data type are often isomorphic.

Commercial relational database management systems relax these comparison restrictions to a greater or lesser degree. For example, you can compare INTEGER and DECIMAL values in commercially available systems because, the reasoning goes, both are numeric types.

Database management systems usually provide internal data type conversion routines to ensure such comparisons can be made. Programming languages generally refer to this as weak typing. The more strict domain comparison rules of the relational model, which are more strongly typed, do not permit these types of comparison to be made. UDTs are strongly typed and do not permit careless comparisons unless you write casts specifically to permit them. See the information about CREATE CAST in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about creating cast functionality for UDTs.

Column Names and Constraints

The topic of domains leads naturally to the topic of naming columns and assigning constraints to them.

Always use a consistent method to define the names of the columns in your databases.

The following set of columns illustrates an unsophisticated example of why it is important to name your columns carefully.

Case Study, Part 1

Suppose you have defined the following two entities:

PrimaryKeyColumn	ColumnA	Date	Date	Date
------------------	---------	------	------	------

PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

PrimaryKeyColumn	Date	Date	Date
PK, SA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

Can you tell if the highlighted dates in these tables all refer to the same point in time? Can you tell if any of them do? If not, then you should not compare them. But how do you know whether the dates are drawn from the same domain or not?

Case Study, Part 2

Now assume that you have identified and named all the domains in your database. The next step is to apply those domains to the identified entities in your logical model.

Consider the same two entities we examined before, only now their columns have been assigned names that represent the domains from which those column values are drawn.

employee				
emp_num	last_name	birth_date	hire_date	termination_date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

order			
order_number	order_date	shipping_date	billing_date
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960

	24 Mar 1960	31 Mar 1960	03 Apr 1960
--	-------------	-------------	-------------

Guidelines for Naming Columns

As it has been defined so far, this column naming convention fails to define the Date domain with enough rigor to prevent an application developer from comparing a shipping date with a birth or hire date, but it provides a sufficient example for illustration as well as a first approximation to how you might define a domain rigorously.

Because ANSI/ISO SQL does not have facilities for defining domains with this level of rigor, your only course of action is to do so textually or to create appropriate distinct user-defined data types (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details about how to create UDTs and their associated database objects). Be aware that you cannot specify constraints on UDT columns, so their usefulness for defining domains is less than it might otherwise be.

You can record these distinctions in several ways.

- DBC.TVFields.CommentString

Create this comment string using the COMMENT SQL statement.

View the comment you create using the Columns system view.

- Column naming conventions
- Metadata repositories
- ATM Domains form

After you have rigorously defined your domains textually, your next step should be to name your columns by qualifying those domain names.

Column Naming Convention

Teradata recommends the following syntax for naming columns:

```
qualifier domain_name
```

qualifier

A unique name to differentiate one use of a domain from all the other uses of that domain.

domain_name

The name of the domain being qualified.

Using this convention, you could define date domains such as the following example names:

- HRDate
- HR_Date

- OrderDate
- Order_Date

The more finely defined domain names naturally lead to tables having columns with names like those in the following tables.

employee				
employee_number	last_name	birth_HR_date	hire_HR_date	term_HR_date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

order			
order_number	order_orddate	shipping_orddate	billing_orddate
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

With this convention, an application programmer would know never to compare an employee birth date with an order shipping date because they do not represent the same thing even though they share the same set of valid values.

A more rigorous approach might be to create distinct user-defined data types on birth_HR_date, hire_HR_date, and term_HR_date, and then create methods that only permit intra-domain comparisons on those domains. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to create UDTs and their associated database objects.

Note, too, that superficially “identical” values are not necessarily drawn from the same domain. The following dollar amounts, for example, do not refer to the same real world objects:

- \$1.00 USD
- \$1.00 Canadian
- \$1.00 Australian

In the physical design phase, you can further limit column values with constraints imposed on them using the CREATE TABLE or ALTER TABLE statements.

For example, the HR_Date domain would probably exclude all weekend and holiday dates, and if your company only ships goods on Fridays, then you would place a Fridays-only constraint on the shipping_orddate column.

Naming Foreign Key Columns

You should also name foreign key columns to match their names in their primary tables. For example, the foreign key columns in the following relation are not only indicated by the name of their primary table (order) but are also demarcated with the characters FK to redundantly indicate their origin in another table.

lineitem		
lineitem_number	order_FK_number	order_FK_year_date
PK	FK	FK

Metadata Definitions

All domain definitions should be stored in a metadata database and those definitions should be made available to all users of the database.

ATM Domains Form

As part of the transition from the logical model of your database to its physical implementation, you should record all domains that you identify on a Domains form. For more information, see [Domains Form](#).

Key Values and Relationships Among Tables

One result of database normalization is the identification of relationships among the tables defined in the database. These relationships are defined on the basis of primary key values shared between tables.

Example: Basic Case

Consider the following basic case. The employee table was defined as a major entity during the logical design phase. Its columns are defined as follows:

employee				
emp_num	emp_last_name	emp_first_name	emp_middle_initial	emp_phone
PK				
...

The employee_phone table was defined as a minor entity, or entity subtype, related to the major entity, or entity supertype, employee table. Its columns are defined as follows:

employee_phone

emp_num_FK_employee	emp_phone_number	emp_phone_comment
PK, FK	PK	
...

Note:

The employee_phone table has a foreign key. That foreign key, emp_num, is the primary key of the employee table. The two tables relate to one another through this primary key relationship.

In this particular relationship, the minor entity, employee_phone, is said to be the Child table because it references another table. That referenced table, employee, is said to be the Parent table in the relationship.

As a sidebar, note that because the referenced column is a primary key, it is by definition uniquely constrained, so it must be defined physically as either of the following two index types.

- Unique primary index, NOT NULL.
- Unique secondary index, NOT NULL.

Such a constraint only needs to be defined physically if the database management system enforces it. If the constraint is defined with a Referential Constraint, Vantage does not enforce the uniqueness on the parent table, so a UPI or USI is not required and the constraint is just assumed to be valid.

Certain database semantics derive from such relationships, and those semantics require a particular type of database constraint to maintain their integrity. You cannot create referential constraints between columns typed as Period, XML, BLOB, or CLOB.

This constraint is referred to as a referential constraint and it is said to maintain referential integrity.

You cannot define database constraints of any kind on columns having the XML, BLOB, or CLOB data types. See [Designing for Database Integrity](#) for more information.

For more information, see [The Referential Integrity Rule](#) and [Domains and Referential Integrity](#).

Domains Form

Lists and describes all the domains for a database.

This form supports the first step in the ATM process (see [Goals of the ATM Process](#)).

Information Recorded in the Domains Form

The Domains form records the following information about each domain:

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this domain pertains.
System	Name of the system on which this domain is defined.

Information Recorded	Description
Domain Name	Name of the domain. This is often the same name as the UDT corresponding to the domain.
Domain Description	Full text description of the domain.
Data Type	Encoded data type for the domain. You must create your own encodings for UDTs if you use them to define your domains.
Max Bytes	Maximum number of bytes a column value drawn from this domain can occupy in disk storage. Note that the number of bytes per character does not necessarily map 1:1 to the number of characters defined for the column by the defined data type. Multibyte character sets are often represented by multiple bytes per single character, and you must account for this information when filling out the Domains form.
Print Format	Representation of the FORMAT clause (if any) used in the definition of columns drawn from this domain.
Constraint #	Constraint number (as recorded on the Constraints form) that applies to this domain, if any.

Data Type Codes

You should also maintain a list of UDTs and the codes you assign to them, staying as close to the existing code naming convention as possible.

The following table lists the valid predefined data types and their DBC.TVFields codes.

Data Type	Code
ARRAY (one-dimensional) VARRAY (one-dimensional) You cannot use any of the predefined UDT data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a one-dimensional ARRAY/VARRAY type.	A1
ARRAY (multidimensional) VARRAY (multidimensional) You cannot use any of the predefined UDT data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a multidimensional ARRAY/VARRAY type.	AN
BINARY LARGE OBJECT (BLOB)	BL(n)
BIGINT	I8
BLOB	BO
BYTE	B(n)

Data Type	Code
BYTEINT	I1
CHARACTER	C(<i>n</i>)
CHARACTER VARYING	CV(<i>n</i>) CV(<i>n</i>) is the same as VC(<i>n</i>)
CLOB	CO
CHARACTER LARGE OBJECT (CLOB)	CL(<i>n</i>)
DATE	D
DECIMAL	DEC (<i>n</i> , <i>m</i>)
DOUBLE PRECISION	DP
FLOAT	F(<i>n</i>)
GRAPHIC	G(<i>n</i>)
INTEGER	I
INTERVAL YEAR	IY
INTERVAL YEAR TO MONTH	IYM
INTERVAL MONTH	IMO
INTERVAL DAY	ID
INTERVAL DAY TO HOUR	IDH
INTERVAL DAY TO MINUTE	IDM
INTERVAL DAY TO SECOND	IDS
INTERVAL HOUR	IH
INTERVAL HOUR TO MINUTE	IHM
INTERVAL HOUR TO SECOND	IHS
INTERVAL MINUTE	IMI
INTERVAL MINUTE TO SECOND	IMS
INTERVAL SECOND	IS
LONG CHARACTER VARYING	LVC
LONG GRAPHIC VARYING	LVG
NUMBER (both exact and approximate forms)	N
NUMERIC	NUM (<i>n</i> , <i>m</i>)

Data Type	Code
PERIOD(DATE) You cannot use any of the predefined Period data types as the base type for creating a distinct UDT, so you cannot create a domain using a distinct UDT based on a PERIOD type.	PD
PERIOD(TIME)	PT
PERIOD(TIME WITH TIME ZONE)	PTTZ
PERIOD(TIMESTAMP)	PTS
PERIOD(TIMESTAMP <UNTIL_CHANGED>) When the ending element value for PERIOD(TIMESTAMP) is UNTIL_CHANGED, the stored value for the ending element is only 1 byte; otherwise it is 10 bytes.	PTS_UC
PERIOD(TIMESTAMP WITH TIME ZONE) When the ending element value for PERIOD(TIMESTAMP) WITH TIME ZONE is UNTIL_CHANGED, the stored value for the ending element is only 1 byte; otherwise, it is 12 bytes.	PTSTZ
PERIOD(TIMESTAMP WITH TIME ZONE <UNTIL_CHANGED>)	PTSTZ_UC
PERIOD (with derived period columns)	PP
REAL	R
SMALLINT	I2
TIME	T
TIMESTAMP	TS
TIME WITH TIME ZONE	TTZ
TIMESTAMP WITH TIME ZONE	TSTZ
VARBYTE	VB(<i>n</i>)
VARCHAR	VC(<i>n</i>) VC(<i>n</i>) is the same as CV(<i>n</i>)
VARCHAR(<i>n</i>) CHARACTER SET GRAPHIC	VG(<i>n</i>)
XML	XML

You should develop your own data type codes for the UDTs your site uses.

Example: Single-Byte Character Set Definitions

This example assumes you are using a single-byte character set to define the Addr, City, and Comment domains.

Domains Page: ____ of: ____				ELDM Page: ____ System: ____	
Domain Name	Domain Description	Data Type	Max Bytes	Print Format	Constraint Number
Addr	Address	CV(30)	30		
Amt	Dollar Amount	D(10,2)	8	\$\$\$\$\$\$9.99	
City	City Name	CV(30)	30		
Comment	Comment	CV(100)	100		
...

Constraints Form

Database constraints and validity checks are sometimes loosely referred to as business rules (see [Semantic Data Integrity Constraints](#) and [Semantic Constraint Specifications](#)). They define conditions that must be met before a given value is permitted to be written to a column such as value ranges, equality or inequality conditions, and intercolumn dependencies. Vantage supports constraints at both the column and table levels. Constraints are defined using the CREATE TABLE and ALTER TABLE SQL statements.

Note:

You cannot define any type of database constraint on columns defined with a UDT, XML, BLOB, or CLOB data type.

Database triggers, too, are often referred to as business rules. They define certain actions that are to be taken when a particular condition occurs during the update of the table on which they are defined.

The Constraints form:

- Lists all constraints, triggers, and validity checks defined for a database.
- Provides input to the Table form.

This form supports the second step in the ATM process (see [Goals of the ATM Process](#)).

Information Recorded on the Constraints Form

The Constraints form records the following information about each constraint, trigger, and validity check.

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this constraint pertains.
System	Name of the business system in which the applications using this constraint are defined.

Information Recorded	Description
Constraint Number	The number that applies to this constraint. This number is used by other forms as a “foreign key” value to refer to the constraints defined by this form.
Constraint Description	Full text description of the constraint.

Constraint Codes

Code	Definition
FK	Foreign key
ID	Identity column
NC	No changes permitted
ND	No duplicates permitted
NN	Not null
PK	Primary key
SA	System-assigned
UA	User-assigned

Example: Constraint Codes

Constraints Page: ____ of: ____	ELDM Page: ____ System: _____
Constraint Number	Constraint Description
001	ND, NN, NC, PK
002	Must be greater than zero (whole positive integer)
003	Mutually exclusive: One required
004	Mutually exclusive: All required
005	No recursion and no loops
006	Excludes Saturdays, Sundays, and legal holidays
007	Prevent delete rule
008	Reassign delete rule
009	Nullify delete rule
010	Cascade delete rule

011	Copy rows to History table before deleting
012	Trigger update to OrderPart Category table on insert to Order or Part tables.
...	...

System Form

A system is a set of application programs, or applications, grouped by business function. Applications are grouped into systems because each cluster of applications typically serves a different user community within the enterprise.

In terms of ATM, an application is a major business function. Each application is implemented by a set of transactions (see [Application Form](#)).

Ensuring that applications are clustered into the appropriate systems is the most important activity undertaken during this stage of the ATM process. The information described by the System form is generic and applies irrespective of the eventual physical implementation of the database.

The System form has two purposes:

- Identify all applications, grouping them by their business system.
- Provide input to the Application and Table forms.

This form supports the third step in the ATM process (see [Goals of the ATM Process](#)).

Information Recorded

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this system pertains.
System Name	Name of the business system in which the applications using this constraint are defined.
System Description	Brief description of the function of the system.
Application ID	Unique identifier for the application being documented.
Application Name	Full text name of the application being documented.
User Contacts	Names and other useful information pertaining to principal users of the application.

Example: System Form

System	ELDM Page: 1 Page: ____ of ____	System: PTS
System Name:	Part Tracking System (PTS)	
System Description:	Tracks current status and history of each part over its lifetime.	

Application ID	Application Name	User Contacts
Cust	Customer tracking	Joy Calder X829
PCat	Part categories	Bill Alpert
Rpt	Parts reporting	Hall Werts
...
...
...

Application Form

The term *transaction* is used very loosely in the context of the Application form. Any of the following activities equates to a transaction.

- Report
- Single-row SELECT query
- Batch load job

The Application form has two purposes.

- Identify the transactions for each application in the system to estimate run frequencies and volumes.
- Provide input to the physical design of the database.

This form supports the fourth step in the ATM process (see [Goals of the ATM Process](#)).

Estimating Run Frequency and Duration

A typical business year has the following characteristics:

Attribute	Count
Number of business days per week	5
Number of hours per business day	8
Number of business weeks	52
Number of business days per year	260
Number of business hours per year	2,080

For each transaction, estimate the typical and peak run frequencies and identify when the action is performed and the duration of its performance.

The task is to estimate parameters for each identified transaction. The parameters to be estimated are the following.

- Normal (nonpeak period) run frequency
- Peak period run frequency
- Point in the business cycle when the peak period occurs
- Duration of the peak period

Example: Transactions for the Order Tracking Application

The following example form models some of the transactions for the Order Tracking application:

Application		ELDM Page: 1		System: PTS		
Application ID: Ord_____		Application Name: Order Entry_____				
Application Description: Generates and tracks part orders_____						
		Run Frequencies				
Transaction ID	Transaction Name	Typical	Peak	When	Length	Total
OrdCl	Close an order	50/day	75/day	EOQ	2 weeks	14 800
			100/day	EOY	4 weeks	
OrdCn	Cancel an order	1/quarter	N/A	N/A	N/A	4
OrdD	Delete closed orders	1/month	N/A	First day of month	N/A	12

Report/Query Analysis Form

The Report/Query Analysis form:

- Captures a simple category of data demographics for each individual transaction within an application within a system:
 - Lists the tables accessed during a transaction.
 - Lists columns used for valued access (if any).
 - Lists columns used for join access (if any).
- Provides input to the Table form.

This form supports the fifth and sixth steps in the ATM process (see [Goals of the ATM Process](#)).

Terminology

Term	Definition
Access column	A column name that would be specified in the WHERE clause of a query. In the following WHERE clause, column_name is the name of the access column: WHERE column_name = 'xyx'
Join column	A column name that would be specified in the ON clause of a join query.

Term	Definition
	<p>In the following ON clause, table_1.column_name and table_2.column_name are the names of join columns:</p> <p>ON table_1.column_name = table_2.column_name</p>

Procedure

Perform the following procedure to complete the Report/Query Analysis form for each transaction:

1. Begin the analysis with standard reports, load requirements, and similar data.
2. Identify the tables involved and enter the names of each on a Report/Query Analysis form.
3. Identify which columns are printed and enter the names of each on the same Report/Query Analysis form that documents their tables.
4. Identify whether rows are accessed by specific column values.
If so, enter the names of each column for which access values must be obtained and mark them with a V.
Note that these must specify equijoin conditions matching on a single value.
5. Identify whether tables must be joined.
If so, enter the names of the join columns and mark them with a J.
Note that these must specify equijoin conditions.
6. Identify the number of rows that qualify for processing. This is not the number of rows in the answer set, but the number of rows that must be processed in order to obtain the answer set.
Use these guidelines to determine what values to use.
 - a. Identify the number of rows, if any, that qualify for value access.
 - b. Identify the number of rows, if any, that qualify for join access.
 - c. Select the smaller of 1 and 2 and use it as the value for the number of rows processed.
 - d. If there are no value access rows and no join access rows, then use the number of rows in the table.

Example: Hospital Database

There are over 1,000 patients in the hospital database used for these examples. Each patient is hospitalized long-term (where *long-term* means \geq one year).

Example: Patient Detail by Room Assignment

The Patient Detail by Room Assignment report derives from the patient and bed tables. Values and selected demographics for those tables are modeled as follows.

Patient		Cardinality: 1,000	
Patient_ID	Bed_Number	Nurse_ID	Doctor_ID
PK	FK, ND	FK	FK

701	A	201	501
702	B	201	502
703	C	202	502
...

Bed	Cardinality: 1,500
Bed_Number	Room_Number
PK, ND	FK
A	101
B	102
C	103
...	...

Patient Detail by Room Assignment			
Room_ Number	Patient_ID	Bed_Number	Nurse_ID
101	701	A	201
	716	F	233
	723	T	205
102	702	B	201
	725	Q	201
	748	L	222
...

Frequency or Importance Ranking: _____			
Report/Query Description: <u>Patient detail by room assignment</u> _____			
Table: Patient		Number of Output Rows: 1,000	
Input Value Or Source	Bed Bed Number		
Access Column(s)	Bed Number		
Table: Bed		Number of Output Rows: 1,500	

Input Value Or Source	Patient Bed Number		
Access Column(s)	Bed Number		
Table:		Number of Output Rows:	
Input Value Or Source			
Access Column(s)			
Table:		Number of Output Rows:	
Input Value Or Source			
Access Column(s)			

Example: Billing Report for Patient

The Billing Report for Patient 705 is more complex than the report analyzed in the previous example. The cardinality of the PatientServiceHistory table is based on the assumption that there is one year of service data kept in the history file, each patient receives an average of two services per day, and there are roughly 1,000 patients in the hospital at any time.

The calculation is done as follows.

$$1,000 \text{ patients} * 365 \text{ days} * 2 \text{ services} = 730,000 \text{ rows.}$$

Service		Cardinality: 125	
Service_ID	Description	Cost	
PK			
801	Appendectomy	\$ 500.00	
802	Tonsillectomy	\$ 350.00	
803	Anesthesia	\$1,000.00	
...	

Patient_Service_History		Cardinality: 730,000	
Patient_ID	Service_ID	Timestamp	
PK			
FK	FK		

701	801	2007-08-05 11:39
701	803	2007-08-05 23:59
705	801	2007-08-05 8:23
705	802	2007-08-05 13:43
705	803	2007-08-05 18:32
705	814	2007-08-05 23:56
...

Billing Report for Patient 705, August 2000				
Date	Service_ID	Description	Cost	
08/05/2007	801	Appendectomy	\$ 500.00	
	802	Tonsillectomy	\$ 350.00	
	803	Anesthesia	\$1,000.00	
	Total for 08/05/2007		\$1,850.00	
08/28/2007	814	Face lift	\$3,200.00	
	Total for 08/28/2007		\$3,200.00	

Example: Summary Report for All Patients on a Particular Date

This report is based on the same tables as that of the previous example. It reports summary information from those tables for all patients on a particular date. The value for the date column is obtained using the EXTRACT function on the Timestamp value for a performed service.

Billing Report for August 5, 2007		
Patient_ID	Number_of_Services	Total_Cost
701	2	\$ 1,500.00
705	4	\$ 2,100.00
713	1	\$ 320.00
783	12	\$ 2,400.00
795	2	\$ 890.00

Frequency or Importance Ranking:

Report/Query Description: <u>Billing report for a specific date</u>			
Table: Patient_Service_History			Number of Output Rows: 2,000
Input Value Or Source	Current_Date	Service Service_ID	
Access Column(s)	Current_Date	Service_ID	
Table: Service			Number of Output Rows: 2
Input Value Or Source	Patient_Service_History Service_ID		
Access Column(s)	Service_ID		
Table:			Number of Output Rows:
Input Value Or Source			
Access Column(s)			
Table:			Number of Output Rows:
Input Value Or Source			
Access Column(s)			

Table Form

The Table form:

- Extends the fully attributed table model created through the logical database design by collecting table- and column-level data demographic information and adding it to the model.
- Provides input to the physical design of the database.

This form introduces data demographics to the ATM process. Supporting analysis requires you to determine various cumulants and then to add that data to information derived from the Column Names and Constraints and Report/Query Analysis forms.

The form supports the seventh, eighth, and ninth steps in the ATM process (see [Goals of the ATM Process](#)).

Approaches to Filling Out the Table Form

Because the determination of data demographics for columns that will be treated as multicolumn indexes is far more complex than the determination of similar data for single column index columns,

the sample analysis and the way you fill out the Table form differs slightly for the single column and multicolumn situations.

The following table points to the Table form topics that apply to the two situations:

Topic	Applies to Single Column Data?	Applies to Multicolumn Data?
Table Form (this topic)	Yes	Yes
Table Form: Basic Information	Yes	Yes
Table Form: Column-Level Information	Yes	Yes
Table Form: Miscellaneous Column-Level Information	Yes	Yes
Table Form: Access Information	Yes	Yes
Table Form: Data Demographics for Single-Column Database Objects	Yes	No
Maximum and Typical Column Value Frequencies	Yes	Yes
Table Form: Data Demographics for Multicolumn Database Objects	No	Yes

Information Collected for the Table Forms

Table form information falls into these broad categories:

- Miscellaneous
- Access
- Data demographics

Miscellaneous Information

- Column name
- Identity column
- Primary key/foreign key
- Constraint number
- Primary index/secondary index

Note the following things about this point.

- This form is filled out during the logical design phase and indexes should not be defined this soon in the process.
- A table might be a nonpartitioned NoPI object, and a table or join index might be a column-partitioned object, in which case it would not have a primary index.
- Sample data

Transcribe this information from the completed Column Names and Constraints form to the Table form.

Access Information

The following Table form information describes value and join accesses:

- Value access frequency
- Join access frequency
- Join access rows

Transcribe this information from the completed Report/Query Analysis form to the Table form.

Data Demographics

The following Table form information describes the demographics of your data:

- Distinct values
- Maximum rows/value
- Rows/null
- Typical rows/value
- Change rating

This information applies only to those columns that have been identified by the Report/Query Analysis activity as taking part in value and join accesses.

The Table form introduces data demographics to the ATM process for the first time. The sources for this information are varied and you might have to use a variety of resources to obtain the data.

Details of this activity are provided in [Table Form: Data Demographics for Single-Column Database Objects](#).

Filling Out the Table Form

Because the Table form is both complicated and extensive, the next several topics provide a brief overview of its components as well as instructions for how to derive the information requested.

Table Form Example

This example is the first page of the Table form for the Patient_Service_History table. A continuation page for this table, which collects data for multicolumn index candidates, is illustrated in [Table Form: Data Demographics for Multicolumn Database Objects](#).

Table Page: of		ELDM Page: System: Patient Tracking	
Table Name: Patient_Service_History	Table Type: History	Cardinality: 730,000	Data Protection: RAID 5
Column Name	Patient_ID	Service_ID	Timestamp
PK/FK/ID	PK		

	FK	FK	
Constraint Number			
Value Access Frequency	1,000	52	365
Join Access Frequency		232	
Join Access Rows		1,400,000	
Distinct Values	1,000	125	365
Maximum Rows/Value	3,650	15,000	4,000
Rows/Null	0	0	0
Typical Rows/Value	720	5840	2,000
Change Rating	0	0	0
PI/SI			
Sample Data	701 702 703	801 802 803	19920101 19920105 19920105

Table Form: Basic Information

This topic explains how to fill out the general information requests named at the top of the Table form.

Sources for the Required Information

The following table points to the sources for the data requested at the top of the Table form.

Information	Source
System	System form
Table name	Entity name from your logical data model
Table type	Entity type from your logical data model
Row count	Varies. <ul style="list-style-type: none"> If an existing electronic database contains the information, query that database for the cardinality of the table in one of two ways. <ul style="list-style-type: none"> Examine current statistics for the table. Execute a simple <code>SELECT COUNT (*)</code> request on the table. If no existing database contains the information, consult with your users to make a best guess cardinality estimate based on estimates of known maxima, not averages.
Data protection	Varied, depending on how your company operates. The following items are all likely sources for this information. <ul style="list-style-type: none"> Users

Information	Source
	<ul style="list-style-type: none"> Enterprise data model IT policies and procedures <p>The following methods for protecting data are available. Note that while you can specify fallback at the table and join index level, RAID protection applies to your entire hardware configuration.</p> <ul style="list-style-type: none"> Fallback Disk I/O integrity checking RAID 1 RAID 5 RAID S Disk I/O integrity checks

Table Form: Column-Level Information

Column-level information derives from several different sources. This topic parses column-level information into three types. Additional topics examine each of those three types in greater detail.

Column-Level Information Types

Column-level information separates into three categories, as indicated by the following table:

Category	Topic to See for More Detailed Information
Miscellaneous	Table Form: Data Demographics for Multicolumn Database Objects
Value and join access	Table Form: Access Information
Data demographics	Table Form: Data Demographics for Single-Column Database Objects

Table Form: Miscellaneous Column-Level Information

This topic explains how to fill out all the column-level information not categorized as either access or data demographics data.

Sources for the Required Information

The following table points to the sources for the miscellaneous column-level data requested by the Table form.

Information	Source
Column name	Column name from your logical data model.
PK/FK/ID	Primary and foreign key designations from your logical data model. Identity column designation identified by this step.

Information	Source
Constraint number	Constraints form.
PI/SI	<p>Primary and secondary index designations identified by this step. Note the following things about this information.</p> <ul style="list-style-type: none"> • This form is filled out during the logical design phase and you should not define indexes this soon in the process. • If you ignore the previous bullet, be aware that neither indexes nor database constraints can be defined on columns that have any of the following data types. <ul style="list-style-type: none"> ◦ ARRAY/VARRAY ◦ BLOB ◦ CLOB ◦ XML ◦ Geospatial ◦ Period ◦ JSON • A table might be a non partitioned NoPI object, and a table or join index might be a column-partitioned object, in which case it would not have a primary index.
Sample data	Sample data from your logical data model.

Table Form: Access Information

This topic explains how to fill out the column access information requests named in the Table form.

Sources for the Required Information

The following table points to the sources for the access data requested by the Table form:

Information	Source
Value access frequency	Report/Query Analysis form
Join access frequency	
Join access rows	

Table Form: Data Demographics for Single-Column Database Objects

This topic explains how to derive column demographics and how to add that information to the Table form.

Capture demographic data only for access and join column sets and primary key column sets.

By collecting this information, you are emulating some of the fundamental data collection activities performed by the COLLECT STATISTICS statement. The data is used for similar purposes as well: to help

you to model your physical database in a way that its performance across all systems and applications is optimal.

Sources for the Required Information

The following table points to the sources for the data demographics requested by the Table form:

Information	Source
Distinct values	<p>Varies.</p> <p>This parameter determines how many unique values (including nulls) exist for the specified column.</p> <ul style="list-style-type: none"> If an existing electronic database contains the information, query that database for the number of distinct values in the column in one of two ways. <p>Examine current statistics for the table.</p> <p>Execute a simple <code>SELECT COUNT DISTINCT</code> request over the required rows.</p> If there is no existing electronic database that contains the information, consult with your users to make a best guess estimate.
Maximum rows /value	<p>Varies.</p> <p>This parameter determines the maximum value for the specified column.</p> <ul style="list-style-type: none"> If there is an existing electronic database that contains the information, query that database for the value that occurs most frequently in the column in one of two ways. <p>Examine current statistics for the table.</p> <p>Execute a simple <code>SELECT MAX column_name</code> request over the required rows.</p> If no existing electronic database contains the information, consult with your users to make a best guess estimate basing your estimate on known maxima, not averages.
Rows/null	<p>This parameter determines how many rows are null for the specified column.</p> <ul style="list-style-type: none"> If there is an existing electronic database that contains the information, query that database for the number of rows having a null in this in one of two ways. <p>Examine current statistics for the table.</p> <p>Execute a simple <code>SELECT COUNT column_name WHERE column_name IS NULL</code> request over the required rows.</p> If no existing electronic database contains the information, consult with your users to make a best guess estimate basing your estimate on known maxima, not averages.
Typical rows/ value	<p>This parameter determines a typical number of rows per value for the specified column. The typical number of rows per value for a column is not necessarily the average value, and the frequency can be so widely dispersed that a reasonably typical value cannot be determined.</p> <p>For information on examples of how to determine a value for this parameter, see Maximum and Typical Column Value Frequencies.</p>
Change rating	<p>Varied, depending on how your company operates. The following items are all likely sources for this information.</p> <ul style="list-style-type: none"> Users Enterprise data model IT policies and procedures <p>The change rating codes are as follows.</p> <ul style="list-style-type: none"> 0 means the data for this column never changes. <p>Examples include primary key columns and columns that contain historical information.</p>

Information	Source
	<ul style="list-style-type: none"> • 1 means the data for this column rarely changes. • 2 - 8 are user-determined and cover anything not covered by codes 0, 1, and 9. • 9 means the data for this column frequently changes.

Maximum and Typical Column Value Frequencies

This topic compares the concepts of maximum value and typical value and indicates methods for estimating the typical value for a column.

Variables are often distributed in such a way that they have no “typical” value, in which case the average, which represents the maximum likelihood value, is the only reasonable choice.

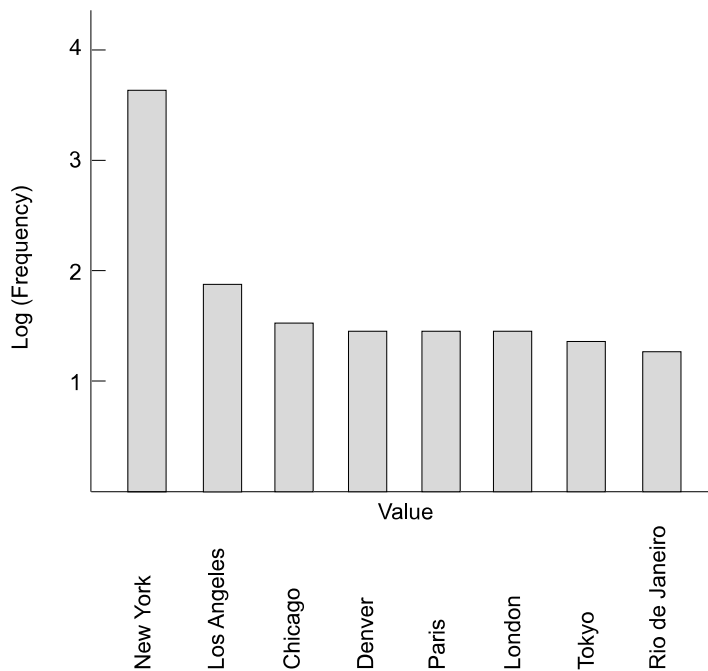
Each of these case studies examines different demographics for the same column. In each case, the data examined is for a City column.

Case Study 1

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602
Los Angeles	80	1.903
Chicago	35	1.544
Denver	30	1.477
Paris	30	1.477
London	30	1.477
Tokyo	25	1.398
Rio de Janeiro	20	1.301

The following histogram graphs the logarithm of the number of rows as a function of row values:



The maximum value for this set is 4,000, but what is the typical value?

It is easy to see by visual inspection of the table above that the typical value for a variable is about 30. Note that the average value for this variable is 531.25, which is by no means typical of the values for the variable.

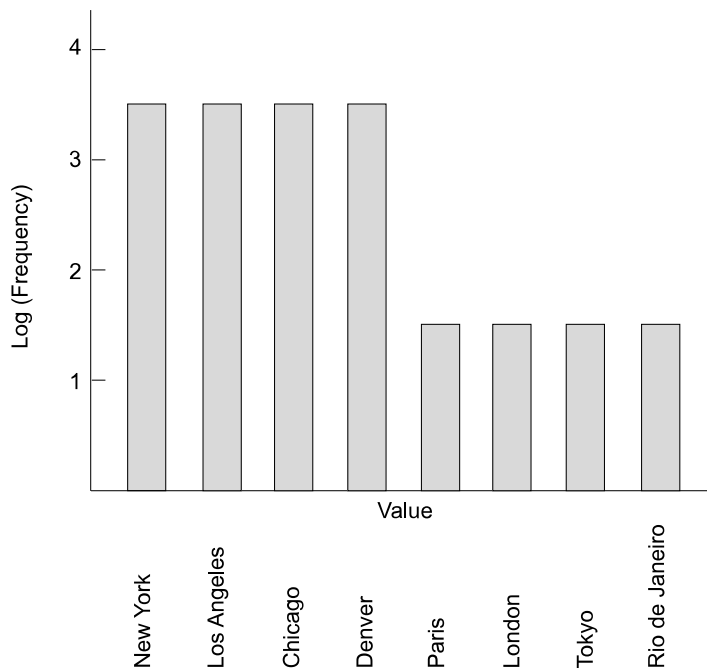
Maximum Value	Typical Value
4,000	30

Case Study 2

The following table indicates the number of distinct occurrences of city column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602
Los Angeles	4,000	3.602
Chicago	4,000	3.602
Denver	4,000	3.602
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values:



The maximum value for this set is 4,000, but what is the typical value?

It is impossible to determine a typical value for the scenario provided by this case history. When you encounter a situation like this, the optimum solution is to use the worst case as your typical value. In this case, that value is 4,000. Note that the average value for this variable is 2,015, which is not only not a typical value for the distribution of the variable, it is never a value for the variable in this case.

Maximum Value	Typical Value
4,000	4,000

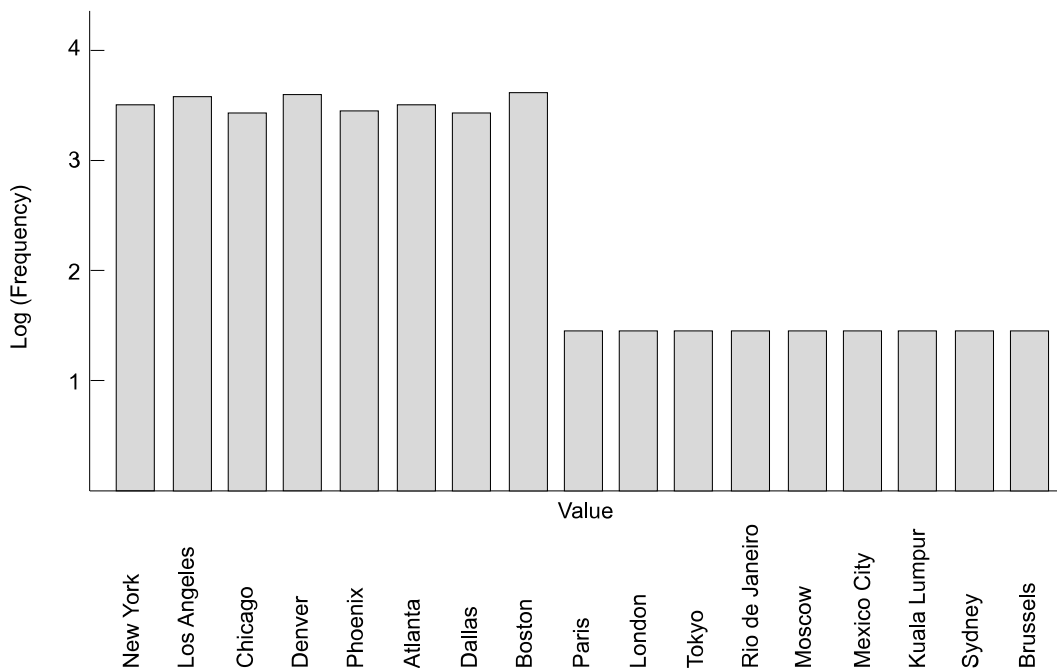
Case Study 3

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4,000	3.602
Los Angeles	4,100	3.613
Chicago	3,800	3.580
Denver	4,200	3.623
Phoenix	3,900	3.591
Atlanta	4,000	3.602

City Value	Frequency of Value	Log Frequency of Value
Dallas	3,800	3.580
Boston	4,150	3.618
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477
Moscow	30	1.477
Mexico City	30	1.477
Kuala Lumpur	30	1.477
Sydney	30	1.477
Brussels	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values:



The maximum value for this set is 4,200, but what is the typical value?

It is impossible to determine an accurate “typical” value for the scenario provided by this case history. Like the scenario presented by [Case Study 2](#), the distribution of values has two peaks at widely diverse points in the distribution. Unlike [Case Study 2](#), the value set clustered around the value 4,000 is not constant.

When you encounter a situation like this, the optimum solution is to use a value around which the largest values cluster as your typical value. In this case, that value is 4,000.

Maximum Value	Typical Value
4,200	4,000

Table Form: Data Demographics for Multicolumn Database Objects

This topic describes how to determine data demographics for multicolumn objects such as composite indexes. Composite indexes are also referred to as multicolumn indexes.

No index of any kind can be based on columns typed as XML, BLOB, or CLOB.

Sources for the Required Information

The following table points to the sources for the multicolumn data demographics requested by the Table form. With the exception of the information documented here, sources are identical to those documented by [Sources for the Required Information](#).

Information	Source
Distinct values	See the following topics. <ul style="list-style-type: none"> • Calculating Multicolumn Demographics • Example Cases for These Guidelines
Maximum rows/value	
Rows/null	
Typical rows/value	

Example: Form for Two Multicolumn Index Candidates

This example form, which continues the form illustrated in [Table Form Example](#) indicates two multicolumn index candidates.

Information acquisition for the second of the two multicolumn specifications is made more simple because the specified columns constitute the primary key for the table.

Because the index candidates are both composite, you cannot use an identity column to define them.

This implies certain demographics by default, as described by the following list. The listed implications hold true irrespective of whether the primary key is uni- or multicolumnar.

- All demographics must be unique.
- No demographics can be null.

Therefore, the value for Rows Null is always 0.

- The value for Distinct Values always matches the Cardinality for the table and the value for Maximum Rows/Value is always 1.

Note that the value for Change Rating is also 0 by default because primary key values should never be changed.

(Multicolumn index considerations) Page: of		ELDM Page: System: Patient Tracking	
Table Name: Patient_Service_History	Table Type: History	Cardinality: 730,000	Data Protection: RAID 5
Column Name	Patient_ID Service_ID	Patient_ID Service_ID Timestamp	
PK/FK/ID	FK	PK	
Constraint Number			
Value Access Frequency		120,000	
Join Access Frequency	2,200	300,000	
Join Access Rows	200,000	300,000	
Distinct Values	8,000	730,000	
Maximum Rows/Value	365	1	
Maximum Rows Null	0	0	
Typical Rows/Value	90	1	
Change Rating	0	0	
PI/SI			

Demographic Trends as a Function of the Number of Columns

Certain characteristics of the demographics of a candidate index change in predictable ways as more columns are added to the index definition.

Selectivity is a measure of the ability of an index to return a highly discriminating subset of rows from a table. The higher the selectivity, the fewer rows retrieved.

- The number of distinct values increases because each additional index column further enhances the singularity of the row.
- The number of rows per value decreases because the number of values increases proportionately.
- The selectivity of the candidate index increases because any set of index column values points to fewer rows.

The following table summarizes these trends:

Index Columns	Number of Distinct Values	Number of Rows Per Value	Selectivity
State	Least	Most	Lowest
State + Zip Code	?	?	?
State + Zip Code + Last Name	Most	Least	Highest

Calculating Multicolumn Demographics

You must have an understanding of the underlying data before you can undertake a legitimate calculation of the demographics for multicolumn situations.

The following table provides some calculation guidelines for determining the number of distinct values in a binary (two column) situation. The guidelines scale as more columns are added to the candidate index.

IF distinct column_1 values ...	THEN the total number of distinct values in the two-column index equals ...	ELSE the number of distinct values is ...
can be paired with distinct values from <i>column_2</i>	the product of the number of distinct values in <i>column_1</i> and the number of distinct values in <i>column_2</i>	approximately equal to the product of the number of distinct values in <i>column_1</i> and the average number of values in <i>column_2</i> that can be paired with a single <i>column_1</i> value.

Example Cases for These Guidelines

The following examples illustrate the guidelines stated in [Calculating Multicolumn Demographics](#).

- Consider the following table fragment. The candidate index for which multicolumn demographics are to be estimated is (ProductCode, ColorCode).

	...	ProductCode	ColorCode	...

The product (ProductCode) * (ColorCode) is true if every product is available in every color.

- Consider the following table fragment. The candidate index for which multicolumn demographics are to be estimated is (StateCode, ZipCode).

	...	StateCode	ZipCode	...

- The product (StateCode) * (ZipCode) is not an accurate estimate of the number of distinct values in the multicolumn index because the resulting product has fewer distinct values than the product of distinct StateCode and distinct ZipCode values.
- If you cannot pair every value in the StateCode column with every value in the ZipCode column, then the number of distinct values in the multicolumn index is approximately equal to (Number of distinct StateCode values) * (Average number of ZipCode values that can be paired with a single StateCode value).

For example, the number of distinct values is the product of the number of states in the US and the average number of zip codes per state, which we estimate to be 20.

Therefore, the number of distinct values for the StateCode + ZipCode index would be roughly $50 * 20$, or 1,000.

- The same considerations hold most of the other rows per value calculations.

For example, if every value of *column_1* can be paired validly with every value for *column_2*, then the result of the division is a fairly accurate estimate.

For each value estimated, the minimum is never less than the number of *column_1* rows per value divided by the number of distinct *column_2* values.

The exception is the count of Rows Null (labeled Rows/Null). See the next section for information on handling multicolumn indexes that are wholly or partially null.

Treating the Rows Null Counts for Composite Indexes

An index can be partly or wholly null. This is almost never a good idea, however. See [Designing for Missing Information](#) for information about SQL nulls and the many problems they cause with database integrity.

Consider the following example:

employee_number	department_number
PK	
FK	FK
103794	7012
?	7012
105378	?
?	?

- The first row has both employee and department numbers.
- The second row has a null employee number and a department number.
- The third row has an employee number and a null department number.
- The fourth row has nulls for both its employee number and its department number.

All four rows are valid primary index rows. Should they all contribute to the count for the Rows Null value or should they be treated differently?

Note:

Only the first of these 4 rows is a valid primary key because by the entity integrity rule (see [Rules for Primary Keys](#)), a primary key cannot contain nulls by definition. This is only one of the many reasons that indexes should not be an issue during logical database design.

The following table provides recommended treatment guidelines for partially and wholly null composite indexes. Composite indexes are also referred to as multicolumn indexes.

IF a composite index value is ...	THEN you should ...
wholly null	add it to the Rows Null count.
partially null	not add it to the Rows Null count.

The Optimizer can detect the number of distinct values from the rows that are partially null and it tracks them separately from the rows that are wholly null. If a column has many nulls, but is not usually specified in predicates, you probably should not include that column in the multicolumn statistics you collect.

For example, suppose you are considering collecting multicolumn statistics on columns 1, 2, and 3 of a table, and their values are like the following, where a QUESTION MARK character represents a null.

	<u>column 1</u>	<u>column 2</u>	<u>column 3</u>
	1	2	3
	?	2	3
	1	?	3
	1	2	?
	?	?	?
	1	2	?
	1	2	?
	1	2	?
	1	2	?
	1	2	?

You would probably not want to collect multicolumn statistics that include column 3 unless it is specified in a large number of predicates.

Row Size Calculation Form

The Row Size Calculation form is used to estimate the size of your tables in bytes. For details about computing row sizes, see [Database-Level Capacity Planning Considerations](#).

Note:

This estimate is of the raw row size without considering all the aspects of physical database design that can affect table size both positively and negatively.

Sources for the Required Information

The following table points to the sources for the miscellaneous column-level data requested by the Row Size Calculation form.

Information	Source
Column name	Column name from your logical data model.
Type	Table form reference to the Constraint form.
Max	See Data Type Considerations .
Avg	For variable length types, must be computed; otherwise, use the value for Max.

Things To Consider When Filling Out the Row Size Calculation Form

Keep the following things in mind as you fill out the Row Size Calculation form:

- The size of the row ID and row header depends on the table partitioning.

Table	Row ID Size (bytes)	Row Header Size (bytes)
not partitioned	8	12
up to 15 partition levels (65,535 combined partitions) 2-bytes used for partition level number in row ID portion of row header	10	14
up to 62 partition levels (more than 65,535 combined partitions) 8-bytes used for partition level number in row ID portion of row header	16	20

These values are true whether the table has a primary index or not.

- BLOB, CLOB, and XML column values are stored outside the row.

The value you should enter for BLOB, CLOB, and XML columns in the Row Size Calculation form is their Object Identifier, which is always 39 bytes.

The size of each XML, BLOB, or CLOB subtable must be calculated separately (for more information, see [Database-Level Capacity Planning Considerations](#)).

- The following system-derived columns consume the following amount of space per row, respectively. Note that the first ROWID can only occur in a join index, and the second two only in result rows.

System-Derived Column Name	Data Type	Number of Bytes Stored for the Internal Information From Which They Are Derived
ROWID	BYTE	<ul style="list-style-type: none"> 10 for 2-byte partitioning or no partitioning 16 for 8-byte partitioning
PARTITION	INTEGER	4 for 2-byte partitioning
	BIGINT	8 for 8-byte partitioning
PARTITION#Ln	INTEGER	4 for 2-byte partitioning
	BIGINT	8 for 8-byte partitioning

- The following system-generated columns consume the following amount of space per row, respectively.

System-Generated Column Type	Data Type	Number of Bytes Stored
Object Identifier (OID)	VARBYTE	39
Identity	Any of the following <ul style="list-style-type: none"> BIGINT BYTEINT DECIMAL(<i>n</i>,0) INTEGER NUMERIC(<i>n</i>,0) NUMBER(<i>n</i>,0) You can only use the fixed form of NUMBER for identity columns. The floating form is not valid. <ul style="list-style-type: none"> SMALLINT 	1 - 8, depending on the data type. This is true even when the DBS Control parameter MaxDecimal is set to 38, because the values generated by an identity column are restricted to a maximum of DECIMAL(18,0) NUMERIC(18,0) or NUMBER(18,0).

- Compressed values and nulls have no effect on the length of their field in a column (they are “stored” as 0 bytes for the field), but can add to the row overhead because of their effect on the number of presence bits in the row header (for more information, see [Presence Bits](#)).

Be aware that each compressed value consumes space in the table header.

- The ending timestamp values for any PERIOD(TIMESTAMP) or PERIOD(TIMESTAMP WITH TIME ZONE) values stored with an UNTIL_CHANGED ending element value consume only 1 byte of storage for their ending element, while PERIOD(TIMESTAMP) and PERIOD(TIMESTAMP WITH TIME ZONE)

values stored with a specified ending element value consume 20 and 24 bytes of storage, respectively, for their ending element.

Procedure

Perform the following procedure to complete the Row Size Calculation form for each table.

1. Identify each column in the table and enter its name in the Column Name column.
2. Identify the data type for each column and enter it in the Type column next to the column name.

IF the type is ...	THEN ...
predefined	use the sizing factors on the right-hand side of the form to determine the size of values stored for each data type.
BLOB, CLOB, or XML	enter 39 bytes for the value of its OID. BLOB, CLOB, and XML values are stored outside of the row in their own subtables, which also must be accounted for in your capacity planning.
a distinct, structured, or ARRAY UDT	use the back of the form to tally its size. When you have identified and tallied all UDT columns in the table, copy their sum into the UDTs item in the sizing factors column on the right-hand side of the form.

3. Determine the maximum value of the specified data type for each column in the table and enter the value in the Max column.
4. Compute the average size of the specified data type for each column in the table and enter the value in the Avg column.
5. Enter the total physical row size, rounded up to an even number of bytes, as Physical Size.

Denormalizing the Physical Schema

This section describes some of the common ways to denormalize the physical implementation of a normalized model. The section also briefly describing the popular technique of dimensional modeling and shows how the most useful attributes of a dimensional model can be emulated for a normalized physical database implementation through the careful use of dimensional views.

The term *denormalization* describes any number of physical implementation techniques that enhance performance by reducing or eliminating the isomorphic mapping of the logical database design on the physical implementation of that design. The result of these operations is usually a violation of the design goal of making databases application-neutral. In other words, a “denormalized” database favors one or a few applications at the expense of all other possible applications.

Strictly speaking, these operations are not denormalization at all. The concept of database schema normalization is logical, not physical. Logical denormalization should be avoided. Develop a normalized design and then, if necessary, adjust the semantic layer of your physical implementation to provide the desired performance enhancement. Finally, use views to tailor the external schema to the usability needs of users and to limit their direct access to base tables (see [Dimensional Views](#)).

Denormalization Issues

The effects of denormalization on database performance are unpredictable: as many applications can be affected negatively by denormalization as are optimized. If you decide to denormalize your database, make sure you always complete your normalized logical model first. Document the pure logical model and keep your documentation of the physical model current as well.

Denormalize the implementation of the logical model only after you have thoroughly analyzed the costs and benefits.

Consider the following list of effects of denormalization before you decide to undertake design changes:

- A denormalized physical implementation can increase hardware costs.

The rows of a denormalized table are always wider than the rows of a fully normalized table. A row cannot span data blocks; therefore, there is a high probability that you will be forced to use a larger data block size for a denormalized table. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.

- While denormalization benefits the applications it is specifically designed to enhance, it often decreases the performance of other applications, thus contravening the goal of maintaining application neutrality for the database.

- A corollary to this observation is the fact that a denormalized database makes it more difficult to implement new, high-performing applications unless the new applications rely on the same denormalized schema components as existing applications.
- Because of the previous two effects, denormalization often increases the cost and complexity of programming.
- Denormalization introduces update anomalies to the database. Remember that the original impetus behind normalization theory was to eliminate update anomalies.

The following graphic uses a simple database to illustrate some common problems encountered with denormalization:

Normalized

Sales Reps

Sales ID	Sales Name	District
----------	------------	----------

Customers

Cust ID	Cust Name	Sales ID
---------	-----------	----------

Invoices

Cust ID	Inv. #	Date	...
---------	--------	------	-----

Denormalized

Sales Reps

Sales ID	Sales Name	District
----------	------------	----------

Customers

Cust ID	Cust Name	Sales ID	Sales Name
---------	-----------	----------	------------

Invoices

Cust ID	Inv. #	Date	Sales Name	District	...
---------	--------	------	------------	----------	-----

Consider the denormalized schema. Notice that the name of the salesman has been duplicated in the Customers and Invoices tables in addition to being in the Sales Reps table, which is its sole location in the normalized form of the database.

This particular denormalization has all of the following impacts:

- When a sales person is reassigned to a different customer, then all accounts represented by that individual must be updated, either individually or by reloading the table with the new sales person added to the accounts in place of the former representative.

Because the Customers, or account, table is relatively small (fewer than a million rows), either method of updating it is a minor cost in most cases.

- At the same time, because the ID and name for the sales person also appear in every Invoice transaction for the account, each transaction in the database must also be updated with the information for the new sales person. This update would probably touch many millions of rows in the Invoice table, and even a reload of the table could easily take several days to complete. This is a very costly operation from any perspective.
- Denormalized rows are always wider rows. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.

Evaluate all these factors carefully before you decide to denormalize large tables. Smaller tables can be denormalized with fewer penalties in those cases where the denormalization significantly improves the performance of frequently performed queries.

Commonly Performed Denormalizations

The following items are typical of the denormalizations that can sometimes be exploited to optimize performance:

- Repeating groups
- Prejoins
- Derived data (fields) and summary tables (column aggregations)

Alternatives to Denormalization

Teradata continues to introduce functions and facilities that permit you to achieve the performance benefits of denormalization while running under a direct physical implementation of your normalized logical model.

Among the available alternatives are the following:

- Views
- Join indexes
- Aggregate join indexes
- Global temporary and volatile tables

Denormalizing with Repeating Groups

Repeating groups are attributes of a non-1NF relation that would be converted to individual tuples in a normalized relation.

Example: Denormalizing with Repeating Groups

For example, this relation has six attributes of sales amounts, one for each of the past six months:

Sales_History						
EmpNum	Sales Figures for Last 6 Months (US Dollars)					
PK	Sales	Sales	Sales	Sales	Sales	Sales
FK						
UPI						
2518	32,389	21,405	18,200	27,200	29,785	35,710

When normalized, the Sales History relation has six tuples that correspond to the same six months of sales expressed by the denormalized relation:

Sales_History		
EmpNum	SalesPeriod	SalesAmount (US Dollars)
PK		
FK		
NUPI		
2518	20011031	32,389
2518	20011130	21,405
2518	20011231	18,200
2518	20010131	27,590
2518	20010228	29,785
2518	20010331	35,710

Reasons to Denormalize With Repeating Groups

The following items are all possible reasons for denormalizing with repeating groups:

- Saves disk space

- Reduces query and load time
- Makes comparisons among values within the repeating group easier
- Many 3GLs and third party query tools work well with this structure

Reasons Not to Denormalize With Repeating Groups

The following items all mitigate the use of repeating groups:

- Makes it difficult to detect which month an attribute corresponds to
- Makes it impossible to compare periods other than months
- Changing the number of columns requires both DDL and application modifications

Denormalizing Through Prejoins

A prejoin moves frequently joined attributes to the same base relation in order to eliminate join processing. Some vendors refer to prejoins as materialized views.

Example: Denormalizing through Prejoins

The following example first indicates two normalized relations, Job and Employee, and then shows how the attributes of the minor relation Job can be carried to the parent relation Employee in order to enhance join processing:

Job

JobCode	JobDesc
PK	NN, ND
UPI	
1015	Programmer
1023	Analyst

Employee

EmpNum	EmpName	JobCode
PK, SA		FK
UPI		
22416	Jones	1023
30547	Smith	1015

This is the denormalized, prejoin form of the same data. This relation violates 2NF:

Employee			
EmpNum	EmpName	JobCode	JobDesc
PK, SA		FK	
UPI			
22416	Jones	1023	Analyst
30547	Smith	1015	Programmer

Reasons to Denormalize Using Prejoins

The following items are all possible reasons for denormalizing with prejoins:

- Performance can be enhanced significantly.
- The method is a good way to handle situations where there are tables having fewer rows than there are AMPs in the configuration.
- The minor entity is retained in the prejoin so anomalies are avoided and data consistency is maintained.

Reasons Not to Denormalize Using Prejoins

You can achieve the same results obtained with prejoins without denormalizing your database schema by using any of the following methods:

- Views with joins (see [Denormalizing Through Views](#))
- Join indexes (see [Denormalizing through Join Indexes](#))
- Global temporary tables (see [Denormalizing Through Global Temporary and Volatile Tables](#))

Denormalizing through Join Indexes

Join indexes provide the performance benefits of prejoin tables without incurring update anomalies and without denormalizing your logical or physical database schemas.

Although join indexes create and manage prejoins and, optionally, aggregates, they do not denormalize the physical implementation of your normalized logical model because they are not a component of the fully normalized physical model.

Remember: normalization is a logical concept, not a physical concept.

Example: Denormalizing through Join Indexes

Consider the prejoin example in [Denormalizing Through Prejoins](#). You can obtain the same performance benefits this denormalization offers without incurring any of its negative effects by creating a join index.

```
CREATE JOIN INDEX EmployeeJob
  AS SELECT (JobCode, JobDescription), (EmployeeNumber, EmployeeName)
  FROM Job JOIN Employee ON JobCode;
```


This join index not only eliminates the possibility for update anomalies, it also reduces storage by row compressing redundant Job table information.

Reasons to Denormalize Using Join Indexes

The following items are all reasons to use join indexes to “denormalize” your database by optimizing join and aggregate processing:

- Update anomalies are eliminated because the system handles all updates to the join index for you, ensuring the integrity of your database.
- Aggregates are also supported for join indexes and can be used to replace base summary tables.

See also [Join and Hash Indexes](#).

Derived Data Attributes

Derived attributes are attributes that are not atomic. Their data can be derived from atomic attributes in the database. Because they are not atomic, they violate the rules of normalization.

Derived attributes fall into these basic types:

- Summary (aggregate) data
- Data that can be directly derived from other attributes

Approaches to Handling Standalone Derived Data

There are occasions when you might want to denormalize standalone calculations for performance reasons. Base the decision to denormalize on the following demographic information, all of which is derived through the ATM process.

- Number of tables and rows involved
- Access frequency
- Data volatility
- Data change schedule

Guidelines for Handling Standalone Derived Data

As a general rule, using an aggregate join index or a global temporary table is preferable to denormalizing the physical implementation of the fully normalized logical model.

The following table provides guidelines on handling standalone derived data attributes by denormalization. The decisions are all based on the demographics of the particular data. When more than one recommended approach is given, and one is preferable to the other, the entries are ranked in order of preference.

Access Frequency	Change Rating	Update Frequency	Recommended Approach
High	High	Dynamic	1. Use an aggregate join index or global temporary table.

Access Frequency	Change Rating	Update Frequency	Recommended Approach
			2. Denormalize the physical implementation of the model.
High	High	Scheduled	Use an aggregate join index or global temporary table.
High	Low	Dynamic	Use an aggregate join index or global temporary table.
High	Low	Scheduled	<ul style="list-style-type: none"> • Use an aggregate join index or global temporary table. • Produce a batch report that calculates the aggregates whenever it is run.
Low	Unknown	Unknown	Calculate the information on demand rather than storing it in the database.

Any time the number of tables and rows involved is small, calculate the derived information on demand.

Reasons Not to Denormalize Using Derived Data

The following items deal with the issues of derived data without denormalizing user base data tables:

- Aggregate join index (see [Aggregate Join Indexes](#))
- Global temporary table with derived column definitions
- View with derived column definitions

Denormalizing Through Global Temporary and Volatile Tables

Global temporary tables have a persistent stored definition just like any base table. The difference is that a global temporary table is materialized only when it is accessed by a DML request for the first time in a session and then remains materialized for the duration of the session unless explicitly dropped. At the close of the session, all rows in the table are dropped. Keep in mind that the containing database or user for a global temporary table uses PERM space to contain the table header on each AMP.

Analogously, volatile tables can have a persistent stored definition if that definition is contained within a macro. When used in this manner, the properties of global temporary and volatile tables are largely identical in regard to persistence of the definition (see the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for other distinctions and differences).

Global temporary tables, like join and hash indexes, are not part of the logical model. Because of this, they can be denormalized to any degree desired, enhancing the performance of targeted applications without affecting the physically implemented normalization of the underlying database schema. The logical model is not affected, but all the benefits of physical schema denormalization are accrued.

It is important to remember that a materialized instance of a global temporary table and a volatile table are local to the session from which they are materialized or created, and only that session can access its materialized instance.

This also means that multiple sessions can simultaneously materialize instances of a global temporary table definition (or volatile tables) that are private to those sessions.

Using Global Temporary Tables and Volatile Tables to Avoid Denormalization

You can use global temporary and volatile tables to avoid the following denormalizations you might otherwise consider:

- Prejoins
- Summary tables and other derived data

This final point is important as an alternative for applications that do not require persistent storage of summary results as offered, for example, by aggregate join indexes.

Using Global Temporary and Volatile Tables to Enhance Performance

You can use global temporary tables to enhance performance in the following ways:

- Simplify application code
- Reduce spool usage
- Eliminate large numbers of joins

This final point is important as an alternative for applications that do not require persistent storage of prejoin results as offered, for example, by join indexes.

Example: Simple Denormalization for Batch Processing

The following global temporary table serves 500 different transactions that create the output it defines. These transactions collectively run over one million times per year, but 95% of them run only on a monthly batch schedule.

With the following table definition stored in the dictionary, the table itself, which violates 2NF, is materialized only when one of those batch transactions accesses it for the first time in a session:

TemporaryBatchOutput

DeptNum	EmpNum	DeptName	LastName	FirstName
PK				
FK	FK			
...

Example: Aggregate Summary Table

The following global temporary table definition, if used infrequently and is not shared, might be an alternative to using an aggregate join index to define the equivalent summary table:

DepartmentAggregations

DeptNum	Period	SumSalary	AvgSalary	EmpCount
PK				
FK				
NUPI				
...

Example: Prejoin

Prejoins are a form of derived relationship among tables. The following table definition, if used infrequently, might be an alternative to using a join index to define the equivalent prejoin table.

This particular table saves the cost of having to join the *Order*, *Location*, and *Customer* tables:

OrderCustomer

OrdNum	CustNum	OrdCost
PK	FK	
FK	NUPI	
...

Denormalizing Through Views

You cannot denormalize a physical database using views, though views can be used to provide the appearance of denormalizing base relations without actually implementing the apparent denormalizations they simulate.

Denormalized views can be a particularly useful solution to the conflicting goals of dimensional and normalized models because it is possible to maintain a normalized physical database while at the same time presenting a virtual multidimensional database to users through the use of a semantic layer based on dimensional views (see "Dimensional Views" below and [Design for Flexible Access Using Views](#)).

Prejoin with Aggregation

The following example creates a prejoin view with aggregation. Note that you can create a functionally identical object as a join index.

```
REPLACE VIEW LargeTableSpaceTotal
  (DBname,Acctname,Tabname,CurrentPermSum,PeakPermSum,  NumVprocs)
AS SELECT DatabaseName,AccountName,TableName,
  SUM (CurrentPerm)(FORMAT '---,---,---,--9'),
  SUM (PeakPerm)(FORMAT '---,---,---,--9'),
```

```

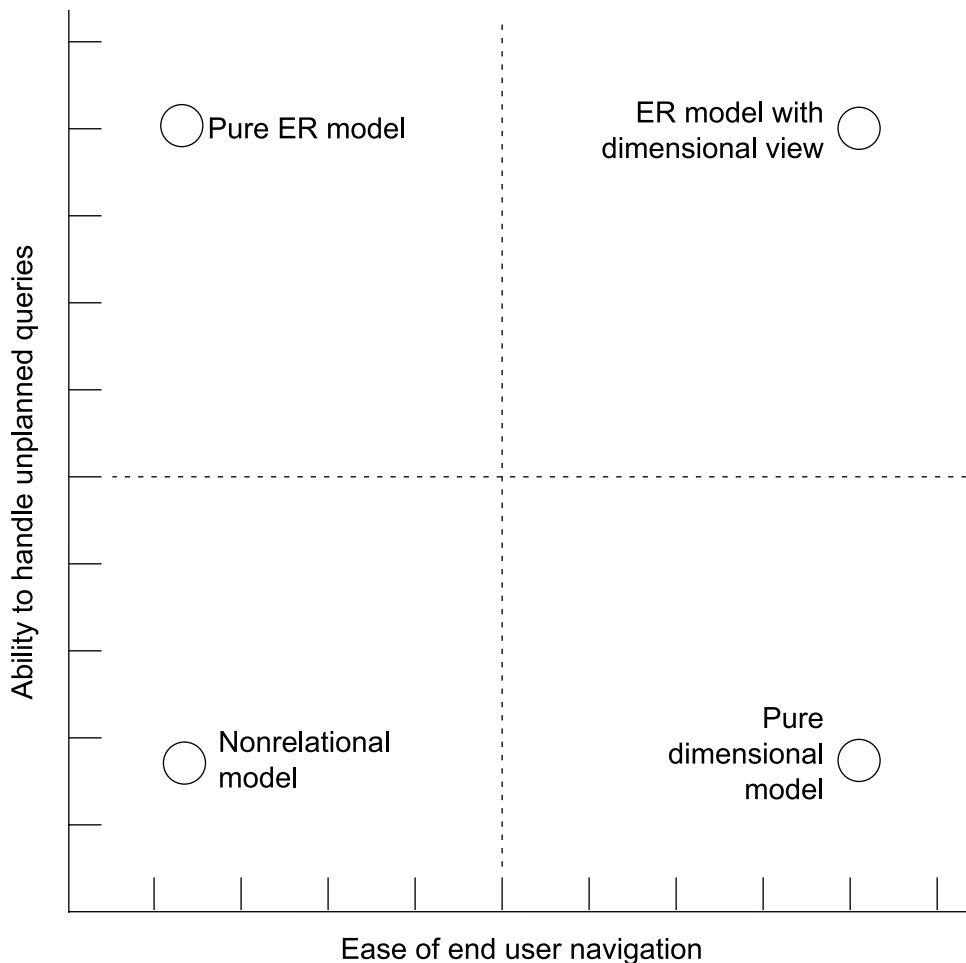
COUNT(*) (FORMAT 'ZZ9')
FROM DBC.TablesizeV
GROUP BY 1, 2, 3
HAVING SUM (currentperm) > 10E9;
SELECT DatabaseName (CHAR(10), TITLE 'DbName'),
       AccountName (CHAR(10), TITLE 'AcctName'),
       TableName (CHAR(16), TITLE 'TableName'), Vproc,
       CurrentPerm (FORMAT '---,---,---,--9'),
       CurrentPerm * 100.0 / CurrentpermSum (AS PctDist, TITLE ' % //
       Distrib', FORMAT 'ZZ9.999'), PctDist * NumVprocs
       (AS PctofAvg, TITLE '% of//AVG ', FORMAT 'ZZ9.9')
FROM LargeTableSpaceTotal, DBC.TablesizeV
WHERE DBname = TablesizeV.DatabaseName
AND AcctName = TablesizeV.AccountName
AND TabName = TablesizeV.TableName
AND PctofAvg > 125.0
ORDER BY 1, 2, 3, 4;

```

Dimensional Views

A dimensional view is a virtual star or snowflake schema layered over detail data maintained in normalized base tables. Not only does such a view provide high performance, but it does so without incurring the update anomalies caused by a physically denormalized database schema.

The following illustration graphs the capability of various data modeling approaches to solve ad hoc and data mining queries as a function of ease-of-navigation of the database. As you can see, a dimensional view of a normalized database optimizes both the capability of the database to handle ad hoc queries and the navigational ease of use desired by many end users.



Many third party reporting and query tools are designed to access data that has been configured in a star schema (see [Dimensional Modeling, Star, and Snowflake Schemas](#)). Dimensional views combine the strengths of the E-R and dimensional models by providing the interface for which these reporting and query tools are optimized.

Access to the data through standard applications, or by unsophisticated end users, can also be accomplished by means of dimensional views. More sophisticated applications, such as ad hoc tactical and strategic queries and data mining explorations can analyze the normalized data either directly or by means of views on the normalized database.

The following procedure outlines a hybrid methodology for developing dimensional views in the context of traditional database design techniques:

1. Develop parallel logical database models.

It makes no difference which model is developed first, nor does it make a difference if the two models are developed in parallel. The order of steps in the following procedure is arbitrary:

- Develop an enterprise E-R model.
- Develop an enterprise DM model.

2. Develop an enterprise physical model based on the E-R model developed in step 1.
3. Implement the physical model designed in step 2.
4. Implement dimensional views to emulate the enterprise DM model developed in step 1 as desired.

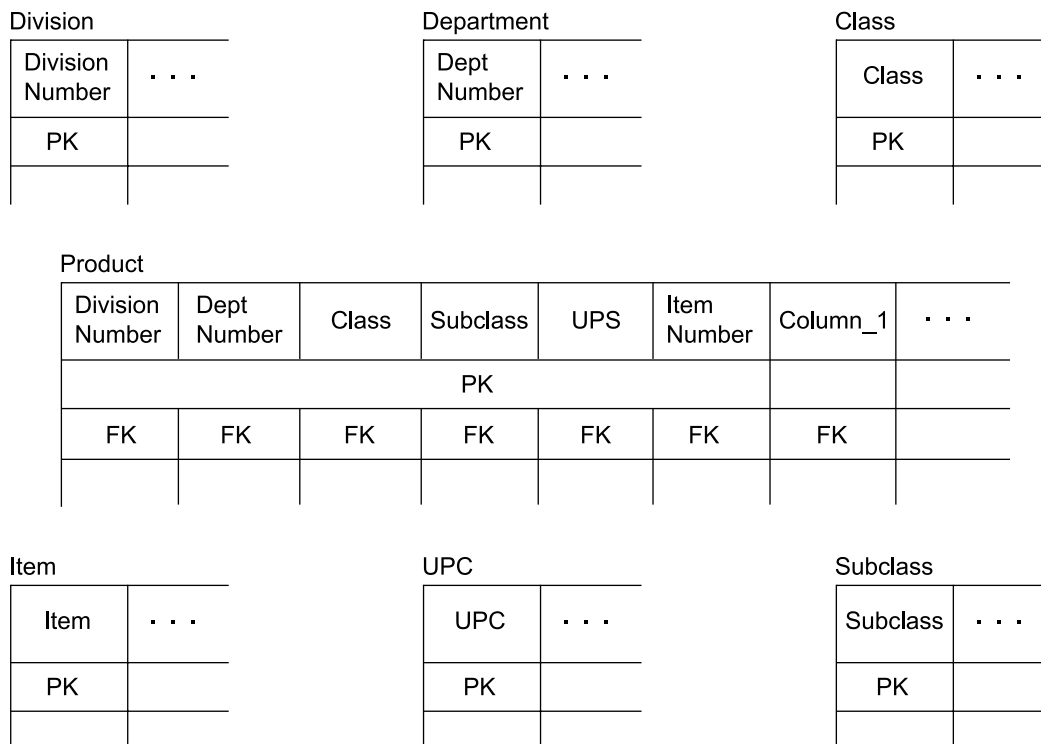
Several Teradata customers use this hybrid methodology to provide a high-performing, flexible design that benefits data manipulation while simultaneously being user- and third-party-tool friendly.

Dimensional Modeling, Star, and Snowflake Schemas

Definition of Dimensional Modeling

Dimensional Modeling is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table.

The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).



Fact Tables and Dimension Tables

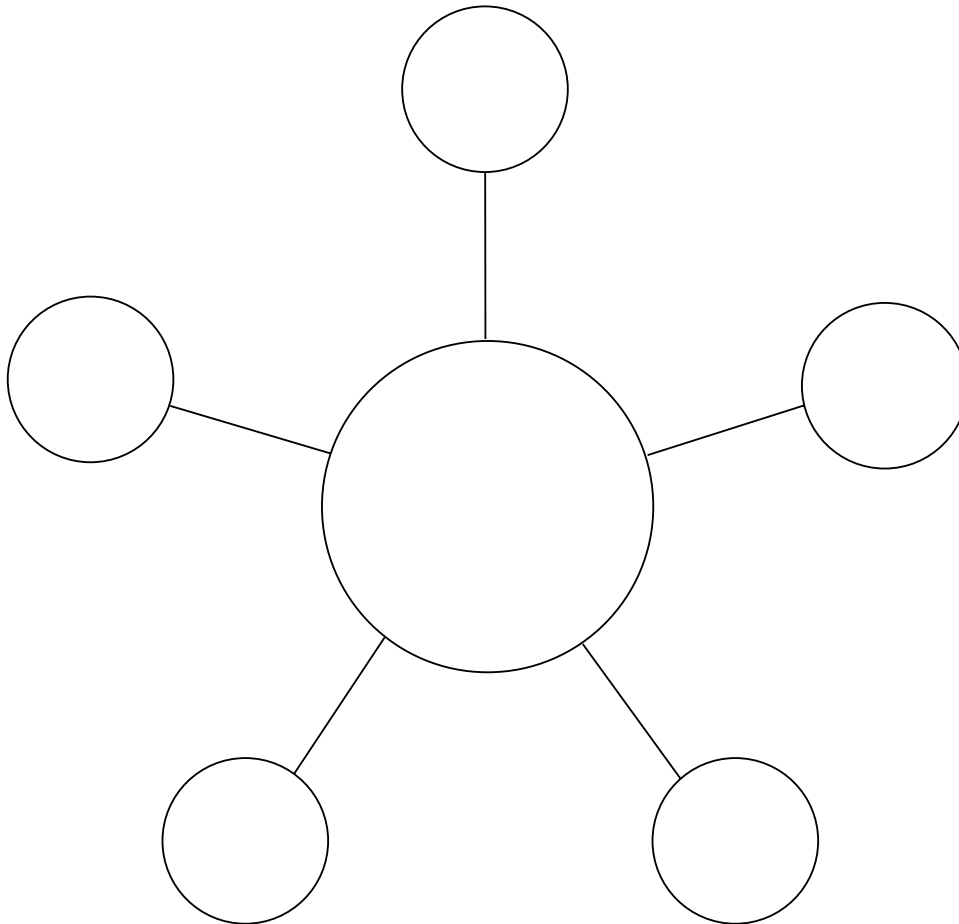
The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the following graphics).

In a dimensional model, fact tables always represent M:M relationships (see [Many-to-Many Relationships](#)). According to the model, a fact table should contain one or more numerical measures (the “facts” of the fact table) that occur for the combination of keys that define each tuple in the table.

Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table.

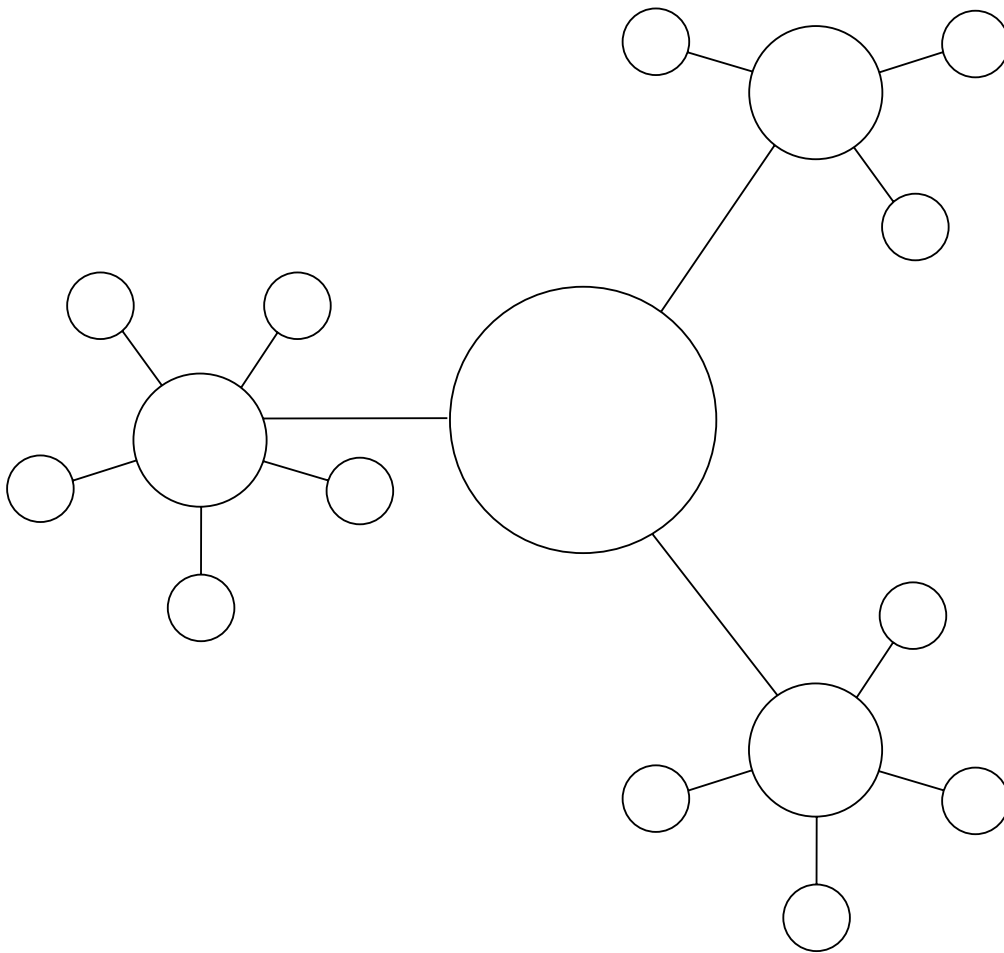
Star Schema

The following graphic illustrates the classic star schema:



Snowflake Schema

The following graphic illustrates the classic snowflake schema:



The E-R Model Versus the DM Model

While a table in a normalized E-R-derived database represents an entity and its relevant atomic descriptors, tables in a DM-derived database represent dimensions of the business rules of the enterprise. The meaning of business rule here is somewhat different from that used by writers in the business rules community, where the term applies to the declarative domain, range, uniqueness, referential, and other constraints you can specify in the database.

While advocates of implementing a normalized physical schema emphasize the flexibility of the model for answering previously undefined questions, DM advocates emphasize its usability because the tables in a DM database are configured in a structure more akin to their business use.

The E-R model for an enterprise is always more complex than a DM model for the same enterprise. While the E-R model might have hundreds of individual relations, the comparable DM model typically has dozens of star join schemas. The dimension tables of the typical DM-derived database are often shared to some extent among the various fact tables in the database.

Indexes and Maps

Indexes can facilitate access to data of interest in a database. Teradata offers several different types of indexes that can help to optimize the performance of your relational databases.

Every table is associated with a map that determines the AMPs to which the table rows are distributed.

Primary Indexes and Primary AMP Indexes

A primary index (PI) or primary AMP index (PA) is an index that, in addition to facilitating access to data, also determines how table rows are distributed among the AMPs of the system. A properly chosen PI or PA can facilitate access to data for common queries, and help ensure that rows are distributed evenly, so each AMP manages a similar amount of data. A PI or PA is defined to use from one to 64 columns of a table for indexing. For each table row, the values of these index columns are combined and hashed. The resulting numeric hash value determines which AMP stores and manages the data in that row. For a PI but not for a PA, the hash value is also used in ordering rows on an AMP. A PA is only allowed for a column-partitioned table.

Data Access Considerations for Choosing a PI or PA

When you choose the columns for a PI or PA, it is important to consider the nature of queries likely to be made against the table. In almost all cases when a query specifies the values for each of the indexed columns, query performance will be faster than the same query if the columns are not indexed.

A PI or PA allows the database to identify the single AMP that stores the data of interest for the specified values of the index, without requiring a costly database search on all the AMPs that store the rows of the table (called a *full-table scan* or FTS). Moreover, for a PI (but not a PA), the rows of interest can be more quickly found on that AMP since the hash value is used in the ordering of the rows on an AMP.

Data Distribution Considerations for Choosing a PI or PA

A *map* associates a unique set of hash values with each of the different AMPs on which the table data is stored. For a PI or PA, the map also determines which AMP receives a particular table row based on a hash value calculated from the values of the index columns in the row.. For more information on maps, see [Maps](#).

When the database receives a request, such as a query or data insertion, each AMP is responsible for processing its portion of table data. The AMPs work in parallel to facilitate processing. If one or more AMPs have significantly more or less data to process than others, the database is said to have a *skewed* data distribution, and performance suffers. Requests cannot be completed until the AMPs with the most data to process have finished their work, reducing the efficiency and benefits of the parallel processing. By choosing an appropriate PI or PA, you can help ensure that table rows are distributed evenly among the AMPs defined by the map used by the table, and take full advantage of Vantage parallel processing.

Unique PIs

A unique primary index (UPI) requires each row in a table to have a unique value for the combined index columns. This type of index often corresponds to the *primary key constraint* or to a *unique constraint* on the table. Because of the uniqueness of the index value for each row, this type of index generally provides even distribution of the table rows.

UPIs can occasionally result in a skewed data distribution if the number of rows in the table is small relative to the number of AMPs in the map used by the table. In this case, you can create a map that contains fewer AMPs and assign it to the table.

Nonunique PIs and PAs

A nonunique primary index (NUPI) or a PA (all PAs are nonunique) neither requires nor enforces uniqueness on the index values. Choosing an inappropriate set of index columns for a NUPI or PA may result in a skewed data distribution so careful consideration of the choice of index columns and the table's map is important.

To enforce uniqueness of the index columns of a NUPI or PA, a separate unique secondary index (USI) can be defined. See [Secondary Indexes](#) for a discussion of USIs.

Joins and Colocation

Tables that use the same map and that have the same columns defining their PIs or PAs can have rows with the same index values distributed to the same AMPs. This fact can be used to help optimize system performance by ensuring tables that are frequently joined have corresponding rows for the join distributed to the same AMPs. Performing a join operation on rows that are located on the same AMPs is faster than joining rows that are stored on different AMPs. When rows to be joined are located on the same AMP, they are said to be *colocated*.

For tables sharing a *sparse map*, the tables must also be defined to have the same *colocation name*. For more information on table colocation, see [Sparse Maps and Table Colocation](#).

Tables with No Primary Index

Teradata recommends that you explicitly specify a PI for most tables without column partitioning and a PA for most column-partitioned tables. However, there are certain types of tables and situations for which a PI or PA is unnecessary and undesirable, due to index processing overhead or an appropriate set of index columns is not known. In these cases, tables can be created as no-primary-index (NoPI) tables.

An example of such tables would be *staging tables*, to which bulk data is batch loaded. Such tables are generally temporary, and, after loading, the data is typically transferred to a PI or PA table using INSERT ... SELECT statements.

To help ensure an even distribution of NoPI table rows among AMPs, the rows (or blocks of rows) are randomly distributed during a load. This helps to assure a relatively even row distribution among the AMPs. For an INSERT ... SELECT statement, rows are locally transferred by default which might cause skew; optionally, a HASH BY clause can be specified to help ensure an even distribution of rows.

Secondary Indexes

Like a PI or a PA, a secondary index (SI) facilitates access to data. An SI can be unique (USI) or nonunique (NUSI). Unlike a PI or a PA, an SI is not used to determine how data is distributed among AMPs. Also, a table may have multiple secondary indexes.

An SI is implemented as a separate *subtable* that is associated with the table (the *base table*) being indexed. SI subtables are maintained by Vantage, and cannot be accessed directly by users after they are created except to be dropped. As data changes in the base table, any associated SI subtables are automatically updated to keep the secondary indexes accurate.

An SI can supplement the benefits of a PI or PA, by indexing additional table columns, thus providing faster data access for additional types of queries. However, because they are separate subtables that must be updated when the data in the base table changes, there is a performance cost to using secondary indexes, especially for tables that change frequently. This maintenance cost must be weighed against the benefit of faster access to SI-indexed data.

Join Indexes

A join index (JI) can facilitate queries that involve joins, aggregations, or access a commonly referenced subset of the data in a table.

Because they are implemented internally as tables, join indexes offer several additional ways to improve efficiency and speed of query processing:

- A join index can be created to pre-join a set of tables.
- A join indexes can be created with aggregation.
- A join index itself can have a PI (or PA if column partitioned), and then the join index rows are distributed to AMPs based on the hashed values of their primary index columns.
- A join index can be partitioned by row, by column, or by a combination of both.
- In addition to limiting the columns of a join index to frequently-joined columns, join indexes can be created using a WHERE clause to limit the rows in the index. If the Optimizer can use the join index in query processing, the query will execute faster if the index has fewer rows than the base tables.

A *multitable join index* pre-joins rows from frequently-joined tables. When queries involve joins that match such a JI, the Teradata Optimizer can plan the query using the pre-joined JI rather than the base tables, speeding up query performance.

A *single-table join index* only references one table. You can create a single-table join index with the same PI or PA definition as another table to which the indexed table is frequently-joined. The join index table rows are then distributed to the same AMPs as the rows of the other table. Consequently, joins between rows of the join index and rows of the joined table are performed locally on each AMP. These joins would not require any row duplication or redistribution. Also, often sorting the rows can be avoided. Thus, join processing is faster. In some situations, a single-table join index can provide better database performance than a multitable join index because the single-table join index requires less maintenance overhead when data changes in the joined tables.

For example, if two frequently joined tables share a PK-FK relationship, a single-table join index can be created for one of the tables, specifying the join index PI as the key columns of that relationship. This will distribute the rows of the join index to the same AMPs as those of the related table, speeding up queries that join the two tables. The single-table join index itself need only be updated when one of the two tables changes.

A single-table join index can also be used to facilitate access to the data of a table by being a commonly referenced subset or aggregation of the data, having a different PI or PA, or a different partitioning scheme. This allows the base table and join index to meet the needs of different types of queries. Note that, for this specific usage of a join index, there is actually no join involved. It is possible that this same join index could be used in other queries to benefit join processing.

Hash Indexes

Hash indexes provide a similar functionality to single-table join indexes. Teradata recommends you use single-table join indexes. Hash indexes are supported only for compatibility with earlier releases of Teradata Database.

Index Considerations

You should evaluate the following considerations when determining which columns to use for an index, and which data types to assign to those columns:

- In order for the Optimizer to choose the most efficient steps for processing a query, it is important that you collect statistics regularly on all indexed columns and on all columns frequently specified in query predicates. This ensures that the Optimizer has the most current information on the database demographics. For more information on the COLLECT STATISTICS (Optimizer Form) statement, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- When there are many rows that have the same hash value (*hash collisions*) for a PI or PA, the AMPs might receive an uneven number of rows. Use the Teradata hash-related functions to check whether PI and PA candidate columns would result in such a skewed row distribution. For more information on hash functions, see [Hash Functions to Evaluate PI and PA Candidates](#).
- Suitability of the data type for indexing:
 -

Numeric data types are often best because they are compact and often tend toward very many distinct values and fewer rows per distinct value, thus leading to fewer hash collisions.

The data type of a column can affect the way column values are hashed. The Teradata hashing algorithm operates on the internal bit pattern representation of the data, rather than on the externally visible data values.

For example, a numeric value stored as a DECIMAL type with precision of one produces a different hash value than the same numeric values stored as DECIMAL with a different precision. An INTEGER type requires only four bytes of storage, but the same number stored as DECIMAL can take up to eight bytes, depending on the size of the number.

- Character data and byte types may be less suitable choices for index columns because they tend not to be compact or, if the length is small, tend to have few distinct values (leading to hash collisions).
- Columns with complex data types (BLOB, CLOB, DATASET, XML, JSON, Period, ARRAY, and VARRAY) cannot be index columns. A column with a geospatial data type can be an index column for a NUSI, if no other columns are included.
- Join and predicate conversions: If joined columns do not have identical data types and precisions, the values from one column must be converted to match the type of the other. This is also true for comparisons made in the WHERE clause. In these cases, the conversion process results in poor query performance reducing the benefits of indexing.

Index Type Comparisons

The following table summarizes the similarities and differences among primary, secondary, join, and hash indexes.

Attribute	Primary Index or Primary AMP Index	Secondary Index	Join Index
Required	No. If a PI, PA, or NoPI is not explicitly defined, Vantage chooses a default PI or NoPI based on other clauses specified and a DBS Control setting.	No	No.
Maximum per table	1	32 for the combined number of secondary and join indexes, including any system defined indexes, which are used to implement PK and UNIQUE constraints. Each multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.	
Maximum number of columns	64	64	64 per referenced base table. No more than 128 columns can be defined for a row compressed join index, 64 each for the fixed and variable parts.
Unique index supported	PI: Yes PA: No	Yes	Yes
Nonunique index supported	Yes	Yes	Yes
Index stored separately, requiring maintenance	No	Yes (as a subtable)	Yes (as an internal table)

Attribute	Primary Index or Primary AMP Index	Secondary Index	Join Index
when base table is updated.			
Partitioning allowed	Yes (with some restrictions)	No	Yes (with some restrictions)
Value ordering allowed	No	Yes for NUSI on a 4-byte or less integer, DECIMAL, and DATE column only.	Yes for a 4-byte or less integer, DECIMAL, and DATE column only (with some restrictions).
Hash ordering	Yes for PI. Hash ordering is on the index columns. No for PA or NoPI.	Yes for NUSI. Hash ordering is on the index columns or specified columns. Yes for USI. Hash ordering is on the index columns.	Yes for a join index with a PI. Hash ordering is on the PI columns. No for a join index with a PA or NoPI.

Data Access Method	Relative Efficiency
UPI	Highly efficient if specific index values are specified in query.
NUPI	Very efficient if specific index values are specified in query, index selectivity is high, and skew is low. For queries of row-partitioned tables, performance degrades as a function of the number of row partitions that must be accessed.
PA	Efficient if specific index values are specified in query and more efficient if PA is based on one column partition that is stored and compressed in COLUMN format.
USI	Very efficient if index values are specified in query.
NUSI	Efficient if index values are specified in query, the number of rows accessed is relatively small compared to the number of rows in the table. Also, A NUSI may be efficient for queries with range conditions or specific values for a subset of the index columns.
JI	Very efficient when index is applicable.
Full-table Scan	Efficient as all AMPs scan their rows in parallel to satisfy a query, without the overhead of additional indexes, or when an index is not applicable.

Data Access Method	AMPs Accessed	Rows Returned	Query Spool Space Required?
UPI	1	0 or 1	No
NUPI	1	0 or more	If a query returns a single response, the response is directly

Data Access Method	AMPs Accessed	Rows Returned	Query Spool Space Required?
			returned and no spool is required, otherwise a spool is required.
PA	1	0 or more	Yes
USI	2 typically. If the base table row and the USI subtable row hash to the same AMP, then only 1 AMP is accessed.	0 or 1	No
NUSI	1 if the NUSI is defined on the same column set as table's PI or PA otherwise, all AMPs in the table's map.	0 or more	Yes
JI	1 if the join index is accessed by its PI or PA, otherwise, all AMPs in the JI's map. Also, additional AMPs may need to be accessed in order to access referenced based table rows.	0 or 1 if the join index is accessed by its UPI, otherwise 0 or more.	If a query returns a single response, the response is directly returned and no spool is required, otherwise a spool is required.
Full-table Scan	All AMPs in the table's map.	0 or more	Yes

Evaluating Indexes

You should evaluate candidate indexes to ensure they are being used by the Optimizer to help satisfy queries more quickly as intended with acceptable overhead. Candidate PIs and PAs should be chosen carefully so that they provide an even row distribution. After you have created indexes, they should be re-evaluated periodically. Teradata provides tools to help you evaluate your indexes.

Hash Functions to Evaluate PI and PA Candidates

Teradata includes system functions that can help you evaluate whether candidate columns for PIs and PAs would result in an even, nonskewed data distribution for the table rows. The following functions are described in more detail in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Function	Description
HASHAMP	Returns the AMP number that corresponds to a given hash bucket number.
HASHBUCKET	Returns the hash bucket number that corresponds to a given row hash value.
HASHROW	Returns the row hash value for an expression or sequence of expressions.

Together, these functions can show how a given set of candidate PI or PA columns will distribute the rows of a table among the AMPs in a map.

Example: Row Distribution of the Employee Table

The following query shows the row distribution of the employee table based on a proposed unique primary index consisting of the single emp_no column. The resulting table allows you to see whether the table rows would be distributed evenly among the table AMPs using this primary index.

```
SELECT HASHAMP(HASHBUCKET(HASHROW(emp_no))), COUNT(*)
FROM employee GROUP BY 1
ORDER BY 1;
```

HASHAMP(HASHBUCKET(HASHROW(emp_no)))	Count(*)
0	561
1	553
2	550
3	545

This is a relatively even distribution among these AMPs, so the emp_no column would make a good unique primary index for this table.

EXPLAIN Request Modifier

You can use the EXPLAIN request modifier with a query to see a textual description of the steps the Optimizer would use to process the query, and determine if your indexes are being used as you intended. For more information about the EXPLAIN request modifier and how to interpret EXPLAIN output, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Note that semantically equivalent queries specified using different syntax can cause the Optimizer to generate different steps to get the same results, and the different paths can have different costs in CPU time. EXPLAIN can help you choose the most efficient form of a query, in addition to ensuring indexes are used.

Other Tools for Evaluating Indexes

The Teradata Index Analyzer tool analyzes index use and can recommend potentially useful secondary indexes. This tool uses the Query Capture Facility (QCF), which captures query activity and, optionally, the Optimizer query execution path steps logged to the Database Query Log (DBQL). Additionally, DBQL logs can be queried directly to determine whether specific indexes are being used in query processing. For more information about Teradata Index Analyzer, QCF, and DBQL, see *Teradata Vantage™ - Database Administration*, B035-1093.

Maps

Under Vantage MAPS Architecture, every table uses a *map* that specifies which AMPs store the rows of the table. The map associates each AMP with a set of hash values. For each row of a table with a PI or

PA, the values of the index columns are combined and hashed. Based on the map, the resulting hash value determines which AMP receives the row.

Tables and join indexes are assigned a map either explicitly or by default when they are created. For tables with fallback, the primary and fallback copies of the row are stored on different AMPs in the map. Secondary index tables use the same map as their base (indexed) table.

Vantage uses two types of maps to track which rows of a table belong on which AMP:

Contiguous map

This is a type of map that includes all AMPs within a specified range. Vantage creates contiguous maps during a system initialization or reconfiguration. Contact Teradata Support Center personnel for help with defining contiguous maps using the Configuration (config) and Reconfiguration (reconfig) utilities.

Sparse map

This is a type of map that includes a subset of AMPs from a contiguous map. Vantage provides two sparse maps you can use, and you can also create new sparse maps if you have appropriate database privileges.

For more information on creating a sparse map, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Use the DBC.MapsVX view to see a listing of the maps that are available to you. Users with appropriate permissions can create new maps.

In the cases of small tables, data distribution using a large contiguous map may be inefficient and introduce potential skew to the system.

- Consider a table that has only 10 rows, and that uses a 1000-AMP contiguous map. A query requiring a full-table scan would require all 1,000 AMPs to attempt to read their rows. The 990 AMPs that store none of the table rows would consume and waste system resources only to determine that they have no rows to process.
- Also, consider a table that has 10000 rows on this map. While rows may be evenly distributed, each AMP would only have about 10 rows; this may be inefficient to have 1,000 AMPs reading a small amount of data and processing it.

In these cases, you can specify in a CREATE or ALTER statement that a small table or join index use a sparse map that includes only a subset of the AMPs in a contiguous map.

Sparse Maps and Table Colocation

Rows from tables that have the same PI or PA and that share the same contiguous map are distributed to the same AMPs. This is called table *colocation*. Colocation provides a performance advantage when tables are joined on their PI or PA columns, because the join processing for corresponding rows happens within the same AMP. This avoids the row replication and redistribution steps that would be required if the corresponding rows from the different tables in the join were stored on different AMPs.

For tables using sparse maps, there is an additional requirement for joins to take advantage of colocation.

To create a sparse map, you specify the parent contiguous map, and the number of AMPs to be included in the new sparse map. The AMPs in the sparse map will be a subset of the AMPs in the parent contiguous map. The specific AMP subset is variable; each table that uses the same sparse map may have its rows distributed to a different subset of AMPs. For example, if many small tables use the same single-AMP sparse map, the data from these tables will not all be distributed to the same AMP. This helps to avoid a skewed data distribution among the AMPs in the parent contiguous map.

Because tables using the same multi-AMP sparse map may not be stored on the same subset of AMPs, even if these tables have the same PI or PA, their corresponding rows would not benefit from colocation during joins. You can force rows of frequently-joined tables to be distributed to the same subset of AMPs by specifying a *colocation name* when you associate the tables with the sparse map. The colocation name forces tables that use the same sparse map to be stored on the same subset of AMPs, providing the same performance improvement for joins that tables using the same contiguous maps have.

Tools for Managing Maps

Some tables remain fixed in size after they are created and loaded, while others may change frequently. A table should use a map that has a number of AMPs appropriate to the table size, leading to an even distribution of the table rows among the AMPs in the map. Each AMP in the map should have a nearly equal number of rows to process. Tables that use contiguous maps are managed by Vantage. Contiguous maps can be changed and created only during a system reconfiguration. However, tables that use sparse maps should be periodically reassessed to ensure they use a map appropriate for the current table size and anticipated growth.

Teradata provides tools to help you manage maps. These tools can help you evaluate whether tables and maps are appropriately matched and move tables to different maps as necessary. They are especially useful when you expand your database system storage and add AMPs because these tools can migrate existing tables to new maps that take advantage of the new storage and processing capacity. You can use these tools to avoid a full system reconfiguration process, which could require a significant maintenance window to take the system offline and redistribute the table rows among the new AMPs.

- Teradata Viewpoint includes portlets that provide a web-based graphical interface to perform many map management tasks. For more information about these portlets, see *Teradata® Viewpoint User Guide*, B035-2206.
- Teradata also provides views and SQL procedures that allow you to manage maps from an SQL command line tool, such as Teradata Studio or BTEQ. The tools consist of two main groups of related procedures:
 - **Advisor** procedures analyze user tables, optionally using information from DBQL-logged query plans, and make recommendations about moving tables to new maps.
 - **Mover** procedures take the output of Advisor procedures and use it to move tables to new maps. This process generates ALTER TABLE statements to assign different maps to tables, and can use multiple concurrent worker sessions to both speed the process and to maintain database availability for other tasks during the changes with a minimal impact on system performance.

For more information on the Advisor and Mover tools and SQL procedures, see *Teradata Vantage™ - Database Administration*, B035-1093 and *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

- To determine what maps are currently defined on the system, query the DBC.MapsV or DBC.MapsVX views. For more information on Data Dictionary views, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

Primary Index, Primary AMP Index, and NoPI Objects

This section describes primary index (PI), primary AMP index (PA), and no primary index (NoPI) objects.

- Primary indexed objects may be partitioned and the index may be unique or nonunique.
The process of selecting a PI is given emphasis in this section. Other topics for primary indexes include unique versus nonunique, partitioned versus nonpartitioned, row distribution, primary index access to rows, various performance considerations, duplicate row checking, and space utilization.
- Primary indexed objects must be column partitioned (with or without row partitioning). A PA is a nonunique index. Row distribution to AMPs and which AMPs to access for a PA are the same as for a PI. Accessing rows (or, more specifically, column partition values) on an AMP for a PA is the same as for a column-partitioned NoPI.
- NoPI objects are tables and join indexes that neither have a primary index nor a primary AMP index. NoPI objects come in two forms.
 - Nonpartitioned NoPI objects
 - Column-partitioned NoPI objects (with or without row partitioning)

This section focuses on user tables, but the considerations for selecting a PI, PA, or NoPI for a table are largely identical to those for selecting a PI, PA, or NoPI for a join index. Any considerations specific to join indexes are documented in [Join and Hash Indexes](#).

Primary Indexes and Primary AMP Indexes

Purposes of a Primary Index or Primary AMP Index

- To define the distribution of the rows to the AMPs.
With the exception of NoPI tables and some join indexes, Vantage distributes table rows across the AMPs based on the hash of their PI or PA value. The choice of columns for the PI or PA affects how even this distribution is. An even distribution of rows to the AMPs is usually critical in picking a PI or PA column set.
- To provide access to rows more efficiently than with a full-table scan.
If the values for all the PI or PA columns are specified in a DML statement, single-AMP access can be made to the rows using that PI or PA value.
With a row-partitioned object, faster access is also possible when values of partitioning columns are specified or if there are constraints on partitioning columns. With a column-partitioned object, only the column partitions with columns needed by the query are accessed.
Other retrievals might use a secondary index, a hash or join index, a full-table scan, or a mix of several different index types.
- To provide for efficient joins.

If there is an equijoin constraint on the PI or PA of a table, it may be possible to do a direct join to the table. That is, rows of the table might not have to be redistributed, spooled, and sorted prior to the join.

- To provide for efficient aggregations.

If the grouping defined by a GROUP BY clause is on the PI or PA of a table, it is often possible to perform a more efficient aggregation.

Restrictions on Primary Indexes and Primary AMP Indexes

- No more than one PI or PA can be defined on a table.

You can also define tables that neither have a PI nor a PA (see [NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes](#) and [Column Partitioning](#)).

- No more than 64 columns can be specified in a PI or PA definition.
- A PI and PA column cannot have an XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, ARRAY, VARRAY, VARIANT_TYPE, Geospatial, or JSON data type.
- A PI and PA column cannot be a row-level security constraint column
- You cannot specify multivalue compression for a PI or PA column

Primary Index Dimensions

Primary indexes are defined in two ways:

- Unique versus nonunique (see [Unique and Nonunique Primary Indexes](#)).
- Partitioned versus nonpartitioned (see [Partitioned and Nonpartitioned Primary Indexes](#)). A partitioned primary index can be partitioned on anywhere from 1 to 62 levels for:
 - Base tables that are not queue tables
 - Global temporary tables
 - Volatile tables
 - Uncompressed join indexes

At most, one level can be column partitioned; the other levels, if any, can be row partitioned. The following types of tables and indexes cannot be partitioned:

- Queue tables
- Global temporary trace tables
- Row-compressed join indexes
- Hash indexes
- Journal tables

See [Partitioned and Nonpartitioned Primary Indexes](#), [Single-Level Partitioning](#), and [Multilevel Partitioning](#).

For hash and join indexes only, the primary index can also be value-ordered.

Primary Index Defaults

You should always explicitly specify a primary index, a primary AMP index, or explicitly specify no primary index for your tables. If you don't, a default is used which might not be appropriate for the object.

Vantage never creates a partitioned PI or NoPI by default and never creates a PA by default

Use the PrimaryIndexDefault field in DBS Control to determine whether Vantage automatically creates a table with a PI or NoPI if the table is created without the PRIMARY INDEX, PRIMARY AMP INDEX, and NO PRIMARY INDEX clause. For more information on the PrimaryIndexDefault field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Problematic Selection of a Primary [AMP] Index Column Set

When the selection of a PI or PA column set is problematic, consider either of the following possible solutions.

- Specify NO PRIMARY INDEX when you create the table.
You can load rows into a NoPI table while you continue to determine an appropriate PI or PA. Such a table is sometimes referred to as a *sandbox* or *staging* table.
- Use an identity column as the PI for the table (see [Surrogate Keys](#)).

Unique and Nonunique Primary Indexes

You can define a primary index to be either unique or nonunique. The choice of the columns to use for the primary index may depend on how its rows are most frequently accessed. The uniqueness of a primary index depends on the whether or not the data in the columns is unique.

This topic describes the differences between the two types as well as their relative advantages and disadvantages.

Unique Primary Indexes

Major entities and subentities are typically assigned unique primary indexes. When a subentity has been defined in the logical model, it typically uses the same unique primary index in the physical model as the major entity it is associated with. This ensures that related rows from the two tables always hash to the same AMP.

Consider the following employee table derived from a major entity in the logical database design. The primary key is also defined as the unique primary index for the table.

employee

employee_ID	employee_name	home_address
PK		
UPI		
6149	Joe Smith	3 Homestead Way
6171	Wei-hee Chan	44 Fifth Avenue
7049	Yuka Maeda	1000 Chestnut Lane
...

Always specify the unique primary index column set as NOT NULL.

Nonunique Primary Indexes

Minor entities are typically assigned nonunique primary indexes defined on the same column as the major entity with which they are associated.

Consider the following employee_phone table derived from a minor entity in the logical database design.

The employee_phone table is related to the employee table by its foreign key, employee_id, which is the primary key of the employee table. Note that the primary index for this table is defined only on the employee_id attribute, which is only half the primary key. This makes the primary index nonunique by definition.

The advantage gained by this is that both employee and employee_phone rows have the same primary index and hash to the same AMP. This means that joins on these tables, which are a frequent occurrence, do not require redistribution of table rows.

You could have defined a NUPI on phone_number because there should not be many duplicate entries for that field (husband-wife-child groupings and roommates being likely examples), but you would lose the advantage of hashing related rows to the same AMPs.

employee_phone

employee_ID	phone_number	phone_remarks
PK		
FK		
NUPI		
6149	555-1234	Home land line. Not after 20:00.
6149	555-9315	Cell phone.
6149	555-8357	Pager.
6171	555-5678	Any time.

employee_ID	phone_number	phone_remarks
7049	555-9012	Never call home number.
...

Employee 6149, Joe Smith, has three different telephone numbers (rows shaded in orange) where he can be reached away from the office.

Polyinstantiation

You can create a USI for a row-level security-protected table as a composite of a row-level security constraint column and the columns of a NUPI for the table. This property can be used to implement polyinstantiation.

Polyinstantiation is a property that enables a relation to contain multiple rows with the same primary key value, where the multiple instances are distinguished by their security levels, where a security level is defined by a row-level security constraint column.

For this property not to violate the relational model, the security level instances would need to be defined as components of a composite primary key.

Partitioned and Nonpartitioned Primary Indexes

A primary index can be either partitioned or nonpartitioned. Primary indexes for join and hash indexes can also either be hash- or value-ordered, while primary indexes for all other table types are only hash-ordered.

The decision to define which of the two choices for a table depends on how its rows are most frequently accessed (see [Single-Level Partitioning Case Studies](#)). This topic describes the differences between the two types as well as their relative advantages and disadvantages.

Nonpartitioned Primary Indexes

When a table is created with a nonpartitioned primary index, its rows are hashed to the appropriate AMPs and stored there in row hash order.

Partitioned Primary Indexes

A partitioned primary index is defined to be either single-level or multilevel depending on how many partitioning expressions are defined for the table.

You can define partitioning for global temporary and volatile tables, for base tables (but not queue tables or hash indexes), and for noncompressed join indexes.

Optimal partitioning expressions are typically coded using CASE_N or RANGE_N expressions based on exact numeric, character or DateTime columns. DateTime expressions can include the BEGIN and END bound functions and the DATE, CURRENT_DATE, and CURRENT_TIMESTAMP functions.

The test value you specify with a `RANGE_N` function must result in a `BYTEINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `DATE`, `TIMESTAMP(n)`, `TIMESTAMP(n) WITH TIME ZONE`, `CHARACTER`, `VARCHAR`, `GRAPHIC`, or `VARCHAR(n) CHARACTER SET GRAPHIC` data type.

Support for the `BEGIN` and `END` bound functions also includes support for the `IS [NOT] UNTIL_CHANGED` and `IS [NOT] UNTIL_CLOSED` functions.

You can specify any valid Date or Time element for the `IS [NOT] UNTIL_CHANGED` and `IS [NOT] UNTIL_CLOSED` functions in partitioning expressions based on the `CASE_N` function.

You cannot specify multivalue compression for the partitioning columns of a partitioning expression.

Row-partitioning optimizes range queries while also providing efficient primary index join strategies. Analyze your range query optimization needs carefully because there are performance tradeoffs between specific range access improvements and possible decrements for primary index accesses and joins and aggregations on the primary index that occur as a function of the number of populated row partitions.

The partitioning maxima using different types of partitioning expressions are listed in the following table. These maxima apply to both row-partitioned tables and join indexes and to column-partitioned tables and join indexes (which may also have row partitioning).

Partitioning Parameter	Maximum Value
Combined partitions for a table with 2-byte partitioning.	65,535
Combined partitions for a table with 8-byte partitioning.	9,223,372,036,854,775,807
Valid range for the number of partitioning levels for a table or join index with 2-byte partitioning.	1 - 15
Valid range for the number of partitioning levels for a table or join index with 8-byte partitioning.	1 - 62
<code>RANGE_N</code> Number of ranges plus <code>NO RANGE [OR UNKNOWN]</code> and <code>UNKNOWN</code> partitions, with 2-byte partitioning.	65,535
<code>RANGE_N</code> Number of ranges plus <code>NO RANGE [OR UNKNOWN]</code> and <code>UNKNOWN</code> partitions (whether or not specified), with 8-byte partitioning.	9,223,372,036,854,775,807
<code>CASE_N</code> Number of conditions.	Bound by the request text size and other limits.

When you have a partitioned table, the bits stored in the internal partition number of the `rowID` represent the combined partition number of the row, which is determined by combining the partitioning expressions of a `PARTITION BY` clause into a single expression. The expression that combines the partitioning expressions of multilevel partitioning into a single combined partition number is called a combined partitioning expression.

The terms internal partition number and combined partition number are defined as follows.

Term	Definition
Internal partition number	<p>A value Vantage calculates from the combined partition number.</p> <p>The internal partition number is used to number partitions internally and is stored in the rowID.</p> <p>The internal and combined partition numbers can be identical if no modification is needed.</p> <p>For single-level partitioning only, partitions for the NO RANGE [OR UNKNOWN], NO CASE [OR UNKNOWN], and UNKNOWN options are placed at fixed internal partitions, followed by partitions for range and conditions following.</p> <p>Modification is required to retain existing internal partition numbers after an ALTER TABLE request that drops or adds ranges or partitions. For a table without partitioning, the internal partition number is always zero.</p>
Combined partition number	<p>The value computed for the partitioning expressions for a row in a partitioned table.</p> <p>For a table without partitioning, the combined partition number is always 0.</p> <p>Combining the results for a table with single-level partitioning, the combined partition number is the same as the value of the single partitioning expression.</p> <p>When you specify the system-determined columns PARTITION or PARTITION#Ln in a request, Vantage converts the internal partition number stored in the rowID into the appropriate combined partition number or the partition number of the specified partitioning level and returns that value to the requestor.</p>

Vantage groups partitioned rows into partition groups on an AMP first by their internal partition number, then by row hash value within each internal partition (if there is a primary index, otherwise by the assigned hash bucket), and then by uniqueness value.

If a SELECT query specifies values for all the primary index columns, the AMP that contains those rows can be determined, and only one AMP needs to be accessed. If the query conditions are not specified on the partitioning columns, then each internal partition can be probed to find the rows based on the hash value, assuming there is no usable alternative index. If conditions are also specified on the partitioning columns, then row partition elimination might further reduce the number of partitions to be probed on that AMP (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about row partition elimination).

If a SELECT query does not specify the values for all the primary index columns, an all-AMP, full-table scan is required for a table with no partitioning when there is no usable alternative index. However, with partitioning, if conditions are specified on the partitioning columns, row partition elimination might reduce what would otherwise be an all-AMP, full-table scan to an all-AMP scan of only the partitions that are not eliminated. The extent of row partition elimination depends on the partitioning expressions, the conditions specified in the query, and the ability of the Optimizer to recognize such opportunities.

Suppose a query only requests orders dated August, 2010. For a nonpartitioned table, the whole table is read to determine which rows are from August because the August rows are scattered throughout the table. For a table partitioned by month, on the other hand, row partition elimination can be used to exclude all the partitions that do not contain activity for August. Because all the August rows are grouped together in the partitioned table, it becomes possible to position directly to the first row for August, then read sequentially until a September row is found, at which point the system stops reading rows. This query requested about 1/12 of the rows so only about 1/12 of the partitioned table must be read. If you partition by day and change the query to make the date August 2nd, an even smaller subset of the table is all that must be read.

When rows for current day activity are inserted later tonight into the nonpartitioned table, they are scattered throughout the entire table, so the average hits per block value is very low. For the partitioned table on the other hand, the inserts are clustered in a smaller area. This provides a better locality of reference. If the partitioning is by week or by day, then the inserts are clustered even more tightly, and the hits per block value is very high.

If you want to delete the oldest month of data, the task is a full-table scan for the nonpartitioned table, but for the partitioned table the rows cluster together, so it is possible to delete them all with a high number of average hits per block. Transient journaling is not done for each of the rows, further enhancing performance when all the rows of a partition are deleted as the last action of a transaction.

You can see that if the table is partitioned on order date, there is no need for a NUSI on order date, and if you had one for the nonpartitioned table, you could drop it when you converted the primary index for the table to a partitioned primary index.

You should consider defining a table with partitioning to support either of the following workload characteristics:

- A majority of the requests in the workload specify range predicates on some column, particularly a date column, of the candidate partitioned table.
- A majority of the queries in the workload specify an equality predicate on some column of the candidate partitioned table, and that column is either of the following:
 - Not the only column in the primary index column set
 - Not a primary index column.

In addition to these two workload characteristics, one of the following sets of characteristics of the primary index should also be considered. One of these three cases is always true:

- The primary index is used for the following applications.
 - Primarily or exclusively to distribute rows evenly across the AMPs.
 - Rarely, if ever, to access rows or to join tables.
 - To access rows using a condition specified on a column that is suitable for partitioning the table.
- The primary index, defined with the entire set of partitioning columns, if any, is used for the following applications.
 - To distribute rows evenly across the AMPs.
 - To access rows directly or as a table join condition.
- The primary index, defined without the entire set of partitioning columns, is used for the following applications.
 - To distribute rows evenly across the AMPs.
 - To access rows directly or as a table join condition.

In this case, you must pay particular attention to weighing and optimizing the performance tradeoffs the situation makes possible.

The following factors characterizes the general performance of partitioned tables with respect to nonpartitioned tables:

- Partitioned access by means of an equality constraint on all primary index and partitioning columns is as efficient as the same access made by means of a nonpartitioned primary index.
- Partitioned access on an equality constraint on the primary index columns and an equality or other constraint on the partitioning columns that limits access to a single partition, produces performance as efficient as that made with a nonpartitioned primary index.
- Partitioned access with a constraint on the partitioning columns can approach that of nonpartitioned primary index access when all the primary index columns are specified with equality constraints on an expression that does not reference any columns, depending on the extent of row partition elimination (an indirect measure of the number of internal partitions that must be probed before the desired row is found) required.
- Partitioned access via an equality condition on the primary index that neither includes all of the partitioning columns nor makes a constraint on the partitioning columns might not be as efficient as access by means of a nonpartitioned primary index depending on the number of internal partitions that contain data, but is generally more efficient than a full-table scan.
- Partitioned access when not all of the primary index columns are specified with equality constraints, but there are conditions specified on the partitioning columns that limit the number of internal partitions that must be scanned is more efficient than access by a nonpartitioned primary index, which requires a full-table scan.

Specifically, these are the major performance characteristics of row-partitioned tables with respect to nonpartitioned tables:

- Inserting a large number of rows can be much faster than the same insert operation into an identical table that it is defined without partitioning.

To optimize load performance, the first partitioning expression of the table preferably should match the row arrival pattern of data, which is most often based on transaction date. In this case, for example, the first partitioning expression should be based on a DATE or TIMESTAMP column.

- Accessing rows using conditions on partitioning columns can be faster than the same access to the same rows when the table is defined with a nonpartitioned primary index.

The performance of SELECT, UPDATE, and DELETE operations is improved when queries specify conditions on the partitioning columns of the table because such predicates cause a higher average number of hits per data block.

The following design considerations are also important for this partitioned performance characteristic:

- The degree of improvement in the performance of select, update, and delete operations is proportional to the extent of row partition elimination that can be realized (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details about row partition elimination).

In other words, the more internal partitions that can be excluded by conditions specified on the partitioning columns for a partitioned table, the better.

- Performance improvements are generally greater as a function of how finely the granularity is defined for the internal partitions.
- Depending on the number of internal partitions that contain data, accessing rows by means of an equality constraint on their primary index can be slower for a partitioned table than for the equivalent

nonpartitioned table when you neither include all of the partitioning columns in the primary index definition nor specify a constraint on those columns in the query.

The following design considerations are important for this partitioned performance characteristic:

- The more coarse the granularity of the partitioning you define, or the more internal partitions eliminated, the less performance degradation you are likely to experience.
- Defining an appropriate secondary index on the partitioned table can sometimes minimize the performance degradation of primary index access.
- Joins can be different for partitioned and nonpartitioned tables that are otherwise equivalent, but the effect of the different join strategies that arise cannot be predicted easily in many cases.

The join plan the Optimizer pursues depends on collected statistics, dynamic-AMP samples, and derived statistics. The usual recommendation applies here: check EXPLAIN reports to determine the best way to design your indexes to achieve the optimal join geography.

The following design considerations are important for this partitioned performance characteristic:

- Primary index-to-primary index joins are more likely to generate different join plans than other partitioned-nonpartitioned join comparisons.

To minimize the potential for performance issues in making primary index-to-primary index joins, consider the following guidelines:

Partition the two tables identically if possible.

A coarser granularity of partitions is likely to be superior to a finer partition granularity.

- Examine your EXPLAIN reports to determine which join methods the Optimizer is selecting to join the tables.

Rowkey-based merge joins are generally better than joins based on another family of join methods.

- Efficient row partition elimination can often convert joins that would otherwise be poor performers into good performers.
- The most likely candidate for poor join performance is found when you are joining a partitioned table with a nonpartitioned table, the partitioned table partitioning column set is not defined in the nonpartitioned table, there are no predicates on the partitioned table partitioning column set, and there are many internal partitions defined for the partitioned table.
- The need for secondary indexes is often different for partitioned and nonpartitioned tables that are otherwise equivalent.

Several opposing scenarios present themselves for resolving the issues that might arise from the existence or absence of secondary indexes:

You probably should ...	IF ...
drop an existing secondary index on the partitioning	it is not required to enforce the uniqueness of the partitioning column set of a partitioning expression. In practice, it is unlikely that uniqueness would be required in this case. This has the following benefits:

You probably should ...	IF ...
column set of a partitioning expression	<ul style="list-style-type: none"> The partitioning itself might provide performance benefits similar to those realized by the secondary index. General system performance is enhanced because there is no need to maintain the secondary index subtable. Not having a secondary index subtable realizes considerable disk savings.
add a new secondary index on the primary index column set	<p>one or more columns of the partitioning column set is not also a member of the primary index column set and the primary index values are unique.</p> <p>In this situation, you must define a USI on the primary index column set to enforce its uniqueness because you cannot define a partitioned primary index to be unique unless its definition contains all of the partitioning columns.</p> <p>In some cases, the addition of the USI results in worse performance.</p>

You must consider all of the following factors when making your analysis of a partitioning expression.

- Would the proposed workloads against the table be better supported by a partitioning expression based on a CASE_N or RANGE_N expression?
- Should the partitioning expression specify a NO CASE, NO CASE OR UNKNOWN, NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN option?

If the test value in a RANGE_N expression can never be null, there is no need for an UNKNOWN or NO RANGE OR UNKNOWN partition.

If a condition in a CASE_N expression can never be unknown, there is no need for an UNKNOWN or NO CASE OR UNKNOWN partition.

- Should the table be partitioned on only one level or on multiple levels?
- If the partitioning expression specifies CURRENT_DATE functions, CURRENT_TIMESTAMP functions, or both, how should the expression be configured to minimize problems deriving from reconciling to a new current date or current timestamp value?
- The query workloads that will be accessing the partitioned table.

This factor must be examined at both the specific, or particular, level and at the general, overall level.

Additional factors to consider are the following:

- Performance
 - Does a nonpartitioned table perform better than a partitioned table for the given workload and for particularly critical queries?
 - Is one partitioning strategy more high-performing than others?
 - Do other indexes such as USIs, NUSIs, join indexes, or hash indexes improve performance?
 - Does a partitioning expression cause significant row partition elimination for queries to occur or not?
- Access methods and predicate conditions

- Is access to the table typically made by primary index, secondary index, or some other access method?
- Do typical queries specify an equality condition on the primary index and include the complete partitioning column set?
- Do typical queries specify a non-equality condition on the primary index or the partitioning columns?
- Join strategies
 - Do typical queries specify an equality condition on the primary index column set (and partitioning column set if they are not identical)?
 - Do typical queries specify an equality condition on the primary index column set but not on the partitioning column set?
 - Do typical query conditions support row partition elimination?
- Data maintenance
 - What are the relative effects of a partitioned table versus a nonpartitioned table with respect to the maintenance workload for the table?
 - If you must define a USI on the primary index to make it unique, how much additional maintenance is required to update the USI subtable?
- Frequency and ease of altering partitions
 - Is a process in place to ensure that ranges are added and dropped as necessary?
 - Does the partitioning expression permit you to add and drop ranges?
 - If the number of rows moved by dropping and adding ranges causes large numbers of rows to be moved, do you have a process in place to instead create a new table with the desired partitioning in place, then INSERT ... SELECT or MERGE the source table rows into the newly created target table with error logging?
 - If you want to delete rows when their partition is dropped, have you specified NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions?
- Backup and restore strategies.

See [Single-Level Partitioning Case Studies](#) for a set of design scenarios that evaluate the performance effects of several different partitioned primary indexes.

See [Single-AMP Queries and Partitioned Tables](#) and [All-AMP Tactical Queries and Partitioned Tables](#) for specific design issues related to partitioned table support for tactical queries.

Partitioning Expression Data Type Considerations

Note:

In addition to the following data type rules and restrictions, be aware that you cannot specify a row-level security constraint column in a partitioning expression.

The partitioning expressions you can define for a partitioned table have certain restrictions regarding the data types you can specify within them and with respect to the data type of the result of the function.

The following table summarizes these restrictions.

Data Type	PARTITION BY		
	RANGE_N	CASE_N	Expression
<ul style="list-style-type: none"> • ARRAY • VARRAY 	N	N	N
BIGINT	Y	X	I
BLOB	N	N	N
BYTE	X	X	X
BYTEINT	Y	X	I
CHARACTER	Y	X	I
CLOB	N	N	N
DATE	Y	X	I
<ul style="list-style-type: none"> • DECIMAL • NUMERIC • NUMBER (exact form) 	X	X	I
<ul style="list-style-type: none"> • DOUBLE PRECISION • FLOAT • REAL • NUMBER (approximate form) 	X	X	I
GRAPHIC	N	X	N
INTEGER	Y	X	Y
INTERVAL YEAR	X	X	I
INTERVAL YEAR TO MONTH	X	X	X
INTERVAL MONTH	X	X	I
INTERVAL DAY	X	X	I
INTERVAL DAY TO HOUR	X	X	X
INTERVAL DAY TO SECOND	X	X	X
INTERVAL SECOND	X	X	X
LONG VARCHAR	Y	X	I
LONG VARCHAR CHARACTER SET GRAPHIC	N	N	N

Data Type	PARTITION BY		
	RANGE_N	CASE_N	Expression
PERIOD The BEGIN and END bound functions are valid in a partitioning expression when they are defined on a valid PERIOD data type column and the result can be cast implicitly to a numeric data type.	N	X	N
SMALLINT	Y	X	I
TIME	X	X	X
TIME WITH TIME ZONE	X	X	X
TIMESTAMP	Y	X	X
TIMESTAMP WITH TIME ZONE	Y	X	X
UDT	N	N	N
VARBYTE	X	X	X
VARCHAR	Y	X	I
VARCHAR(n) CHARACTER SET GRAPHIC	N	N	N
<ul style="list-style-type: none"> XML XMLTYPE 	N	N	N

The following table explains the abbreviations used in the previous table.

Symbol	Definition
I	Valid for a partitioning expression. If the type is also the data type of the result, then it must be such that it can be cast to a valid INTEGER or BIGINT value.
N	Not valid for a partitioning expression. If the partitioning expression is defined using a CASE_N function, then this type is not valid for the CASE_N condition.
X	Valid for a partitioning expression, but cannot be the data type of the result of the expression. If the partitioning expression is defined using a CASE_N function, then this type is valid for the CASE_N condition.
Y	Valid for a partitioning expression and valid as the data type of the result of the partitioning expression.

Column Partitioning a Table or Join Index

Besides partitioning tables or join indexes on their rows, you can also partition them on their columns. Column partitioning enables Teradata Columnar.

Column partitioning is a physical database design choice that is not suitable for all workloads. For example, column partitioning is usually not suitable for workloads that often select both a significant number of rows and project many columns from a table. However, column partitioning might be suitable if a request selects a significant number of rows, but projects only a few columns, or conversely, if a request projects many columns, but only selects a small number of rows.

Column partitioning can be suitable for the case where both a small number of rows are selected and only a few columns are projected. See [Column Partitioning](#) for more information about how table and join indexes can be partitioned on their columns and how column-partitioned tables and join indexes can be applied optimally.

A column-partitioned table or join index can also have 1 or more row partitioning levels.

Choosing an Indexing Method for a Column-Partitioned Table or Join Index

Column-partitioned tables and join indexes can have one of these indexing methods:

- Primary AMP index (PA)
- Primary index (PI)
- No primary index (NoPI)

NoPI is covered in the following topic [NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes](#).

Benefits and Considerations for Column-Partitioned Tables with a Primary AMP Index

Column-partitioned tables with a primary AMP index provide these benefits:

- Single-AMP access
- Local access when joining on the primary AMP index columns
- Better performance for aggregations than column-partitioned NoPI tables when grouping by the primary AMP index columns.
- More efficient joins than column-partitioned NoPI tables:
 - Allows a local dynamic hash or product join when performing an equality join on primary AMP index columns and the other relation has the same primary AMP index or PI columns.
 - Allows a dynamic hash or product join with redistribution instead of duplication of the other relation when performing an equality join on primary AMP index columns and the other relation does not have the same primary AMP index or PI columns.
 - Reduces memory usage, CPU, and I/O.
 - Allows for the other relation to be much larger.

Customers may be interested in this indexing method to replace already existing column-partitioned NoPI tables or if they are interested in implementing column-partitioning for the first time.

Considerations include the following:

- Load times may increase. INSERT-SELECT requires row redistribution if the source does not have the same primary AMP index or PI columns.
- Row header compression and autocompression are similar to that on column-partitioned NoPI tables.
- A column-partitioned primary AMP index may not be defined as UNIQUE, but it can have a USI or primary key on the same columns.
- Other limitations apply to all column-partitioned tables. Column-partitioning is not supported for set tables, queue tables, global temporary tables, volatile tables, derived tables, multitable or aggregate join indexes, compressed join indexes, hash indexes, or secondary indexes.

Benefits and Considerations for Column-Partitioned Tables with a Primary Index

Column-partitioned tables with a primary index provide these benefits:

- Single-AMP access
- More efficient aggregations than NoPI
- Direct merge hash (and rowkey) join to table when a join is on the PI columns

Customers may be interested in this indexing method if they want to implement only a few partitions, for example, to vertically partition a PI table into frequently used columns and rarely used columns. Another use may be to partition rows frequently used in WHERE and SELECT clauses. This method provides the advantages of a PI table but reduces the amount of data read for most requests.

Considerations include the following:

- Load times may increase. INSERT-SELECT requires row redistribution and sort by combined partition/hash value of the source relation if the source relation does not have the same PI columns.
- Row header compression and autocompression may be less effective than using column-partitioned primary AMP index or column-partitioned NoPI. In most cases, the column-partitioned table with a PI will be larger than the table without column partitioning.
- The number of table headers increases (compared with column-partitioned NoPI tables), but for wide rows and many rows per value the effect is negligible. The increase in row headers may become excessive as you add more column partitions, especially if the PI is unique or nearly unique.
- Sliding window joins are not supported.
- Other limitations apply to all column-partitioned tables. Column-partitioning is not supported for set tables, queue tables, global temporary tables, volatile tables, derived tables, multitable or aggregate join indexes, compressed join indexes, hash indexes, or secondary indexes.

NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes

A NoPI object is a table or join index that does not have a primary index or a primary AMP index and always has a table kind of MULTISSET.

The basic types of NoPI objects are:

- Nonpartitioned NoPI tables
- Column-partitioned NoPI tables and NoPI join indexes (these may also have row partitioning)

The chief purpose of nonpartitioned NoPI tables is as staging or sandbox tables. FastLoad can efficiently load data into empty nonpartitioned NoPI staging tables because NoPI tables do not have the overhead of row distribution among the AMPs and sorting the rows on the AMPs by rowhash.

Nonpartitioned NoPI tables are also critical to support Extended MultiLoad Protocol (MLOADX). A nonpartitioned NoPI staging table is used for each MLOADX job.

The optimal method of loading rows into any type of column-partitioned table from an external client is to use FastLoad to insert the rows into a staging table, then use an INSERT ... SELECT request to load the rows from the source staging table into the column-partitioned target table.

You can also use Teradata Parallel Data Pump array INSERT operations to load rows into a column-partitioned table.

Because there is no primary index or a primary AMP index for the rows of a NoPI table, its rows are not hashed to an AMP based on their primary index or a primary AMP index value. Instead, Vantage either hashes on the Query ID for a row, or it uses a different algorithm to assign the row to its home AMP.

Vantage then generates a RowID for each row in a NoPI table by using a hash bucket that an AMP owns. This strategy makes fallback and index maintenance very similar to their maintenance on a PI table.

Global temporary tables and volatile tables can be defined as nonpartitioned NoPI tables but not as partitioned NoPI tables.

Column-partitioned tables and column-partitioned join indexes can be defined without a primary index but can also be defined with a primary index or a primary AMP index. See [Column Partitioning](#) for details about column partitioning and NoPI tables and join indexes.

INSERT ... SELECT into NoPI Tables

When the target table of an INSERT ... SELECT request is a NoPI table, Vantage inserts the data from the source table locally into the target table, whether it comes directly from the source table or from an intermediate spool. This is very efficient because it avoids a redistribution and sort. However, if the source table or the resulting spool is skewed, the target table can also be skewed. In this case, you can specify a HASH BY clause to redistribute the data from the source before Vantage executes the local copy operation.

Consider using hash expressions that provide good distribution and, if appropriate, improve the effectiveness of autocompression for the insertion of rows into the target table. Alternatively, you can specify HASH BY RANDOM to achieve good distribution if there is not a clear choice for the expressions to hash on.

When inserting into a column-partitioned NoPI table, also consider specifying a LOCAL ORDER BY clause with the INSERT ... SELECT request to improve the effectiveness of autocompression.

Uses for Nonpartitioned NoPI Tables

Nonpartitioned NoPI tables are particularly useful as staging tables for bulk data loads. When a table has no primary index or a primary AMP index, its rows can be dispatched to any given AMP arbitrarily and the rows do not need to be sorted, so the system can load data into a staging table faster and more efficiently using FastLoad or Teradata Parallel Data Pump array INSERT operations. You can only use FastLoad to load rows into a NoPI table when it is unpopulated, not partitioned, and there are no USIs.

You must use Teradata Parallel Data Pump array INSERT operations to load rows into NoPI tables that are already populated. If a NoPI table is defined with a USI, Vantage checks for an already existing row with the same value for the USI column (to prevent duplicate rows) when you use Teradata Parallel Data Pump array INSERT operations to insert rows into it.

By storing bulk loaded rows on any arbitrary AMP, the performance impact for both CPU and I/O is reduced significantly. After having been received by Vantage, all of the rows can be appended to a nonpartitioned or column-partitioned NoPI table without needing to be redistributed to their hash-owning AMPs.

Because there is no requirement for such tables to maintain their rows in any particular order, the system need not sort them. The performance advantage realized from NoPI tables is achieved optimally for applications that load data into a staging table, which must first undergo a conversion to some other form, and then be redistributed before they are stored in a secondary staging table or the target table.

Using a nonpartitioned NoPI table as a staging table for such applications avoids the row redistribution and sorting required for primary-indexed staging tables. Another advantage of nonpartitioned NoPI tables is that you can quickly load data into them and be finished with the acquisition phase of the utility operation, which frees client resources for other applications.

Both NoPI and column-partitioned NoPI tables are also useful as so-called sandbox tables when an appropriate primary index has not yet been defined for the primary-indexed table they will eventually populate. This use of a NoPI table enables you to experiment with several different primary index possibilities before deciding on the most optimal choice for your particular application workloads.

Rules and Limitations for NoPI and Column-Partitioned Tables

The rules and limitations for NoPI tables and any type of column-partitioned table are the same as those for primary-indexed tables with the following exceptions:

- You cannot create a nonpartitioned NoPI join index.
You can create a column-partitioned join index (with or without row partitioning).
- You cannot create a NoPI or column-partitioned:
 - Queue table
 - Error table
 - SET table

The default table type for NoPI and column-partitioned tables in both Teradata and ANSI/ISO session modes is always MULTiset.

- Global temporary trace tables

Global temporary trace tables do not have a primary index or a primary AMP index by default; however, you are not allowed to specify the NO PRIMARY INDEX option when you create a global temporary table.

- If none of the clauses PRIMARY INDEX (*column_list*), PRIMARY AMP INDEX, NO PRIMARY INDEX, or PARTITION BY are specified explicitly in a CREATE TABLE or CREATE JOIN INDEX request, whether the table or join index is created with or without a primary index or primary AMP index generally depends on whether a PRIMARY KEY or UNIQUE constraint is specified for any of the columns and on the setting of the DBS Control field PrimaryIndexDefault (see [Primary Index Defaults](#) and *Teradata Vantage™ - Database Utilities*, B035-1102 for details and exceptions).
- Neither NoPI tables nor column-partitioned tables can specify a permanent journal.
- Nonpartitioned NoPI tables cannot specify an identity column.

Column-partitioned tables can specify an identity column.

- Hash indexes cannot be defined on NoPI or column-partitioned tables.
- SQL UPDATE (Upsert Form) requests cannot update either a NoPI or a column-partitioned target table.
- SQL MERGE requests cannot update or insert into either a NoPI or a column-partitioned target table.
- You cannot load rows into either a nonpartitioned NoPI or a column-partitioned table using the MultiLoad utility.

You can define all of the following commonly used features for both NoPI and column-partitioned tables:

- Fallback
- Secondary indexes
- Join indexes
- PRIMARY KEY and UNIQUE column constraints
- CHECK constraints
- FOREIGN KEY constraints
- Triggers
- XML, BLOB, and CLOB columns.

Note:

Because there is normally only one row hash value per AMP for NoPI tables, there is also a limit of approximately 256M rows per AMP for NoPI tables that contain columns typed as XML, BLOB, or CLOB.

You can define any of the following table types as NoPI tables:

- Nonpartitioned base tables
- Column-partitioned base tables (with or without row partitioning)

- Column-partitioned, single-table, non-aggregate, noncompressed join indexes (with or without row partitioning)
- Nonpartitioned global temporary tables
- Nonpartitioned volatile tables

Manipulating Nonpartitioned NoPI Table Rows

After a nonpartitioned NoPI staging table has been populated with rows, you can execute one of the following types of DML request to move the nonpartitioned NoPI staging table source rows to a target table.

- INSERT ... SELECT
- MERGE (for a primary-indexed target table only)
- UPDATE FROM (for a primary-indexed target table only)

For these cases with a primary-indexed target table, Vantage reads the rows in the nonpartitioned NoPI source table and then redistributes them in the same way it redistributes them from a primary-indexed table to their hash-owning AMPs based on the primary index of the target table. For a NoPI target table, the rows are locally copied, unless the INSERT ... SELECT specifies a HASH BY clause.

You can use the following DML statements to manipulate or retrieve NoPI table rows prior to moving them to their target table.

- DELETE
- INSERT
- SELECT
- UPDATE

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details of the limitations of these statements when they are used with NoPI tables.

You cannot use UPDATE (Upsert Form) requests to change data in either a NoPI or a column-partitioned table.

Without a primary index or a primary AMP index, there can be no single-AMP access to table rows. However, you can create both unique and nonunique secondary indexes (see [Secondary Indexes](#)) and join indexes (see [Join and Hash Indexes](#)) on NoPI tables, and with appropriate secondary indexes defined, you can specify those indexes in your query conditions in such a way as to avoid full-table scans. You cannot specify a join index in a query condition, but if an appropriate join index exists, the Optimizer can use it to create the access or join plan for a request.

You cannot use the MultiLoad utility to load rows into either NoPI or column-partitioned tables.

You can use FastLoad to load rows into an empty nonpartitioned NoPI table, but not to load rows into a column-partitioned table.

Keep in mind that secondary and join indexes can slow the loading of rows into a NoPI table using Teradata Parallel Data Load array INSERTs. FastLoad is more efficient than Teradata Parallel Data Pump for loading rows into an empty nonpartitioned NoPI table because Vantage processes each Teradata Parallel Data Pump request as a separate transaction.

If you use FastLoad to load rows into a nonpartitioned NoPI table, you cannot create any secondary or join indexes, CHECK constraints, referential integrity constraints, or triggers on the table until after the load operation has completed because FastLoad cannot load rows into a table defined with any of those features. You can use FastLoad to load rows into an empty nonpartitioned NoPI tables that is defined with row-level security constraints, however.

Related Information

Topic	Reference
Column-partitioned tables and join indexes	Column Partitioning
Using both NoPI and column-partitioned tables	CREATE TABLE information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184
NoPI tables	<i>No Primary Index (NoPI) Table User Guide Orange Book</i> , 541-0007565
Column-partitioned tables and join indexes	<i>Teradata® Columnar Primer Orange Book</i> , TDN0009884

Column Partitioning

This topic describes the structure of column-partitioned tables and join indexes.

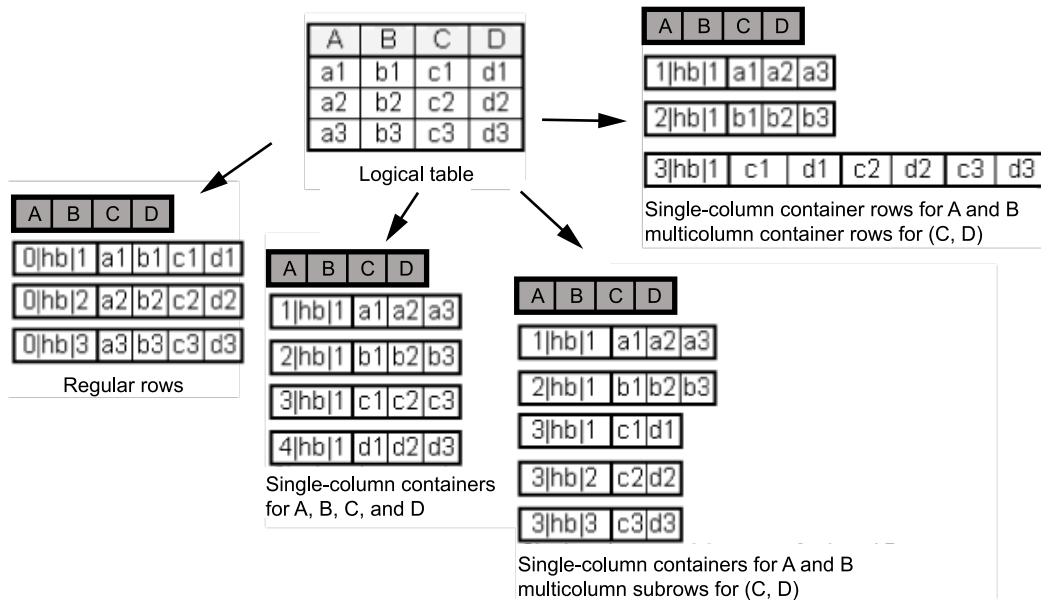
When autocompression methods are described, they are described only to indicate how their use affects the content and size of the row header. Because Vantage determines whether to use autocompression and which autocompression methods to use for a container, they are not documented any further in the Teradata Vantage documentation library.

Options for Storing Data on an AMP

The following graphic shows four options for storing a table in an AMP, where the darkened row for each table represents a table header, one on data for each AMP, for a physical table. The table header defines the layout and other information about the physical table.

Moving from left to right, the structures represent the layouts of four different methods of storage.

- Traditional row storage without partitioning
- A column-partitioned table with single-column containers (COLUMN format), one for each column
- A column-partitioned table with single-column containers for columns A and B; multicolumn subrows (ROW format) for columns C and D
- A column-partitioned table with single-column containers for columns A and B; multicolumn containers for columns C and D



About Column-Partitioned Tables

If a request requires multiple columns to return the requested result set, the Optimizer query plan includes assembling projected column values from selected table rows together to form the result rows. This can be combined with row partition elimination to further reduce the data that must be accessed to satisfy a query.

Column partitioning has various system-applied compression techniques available to it that can be employed automatically to reduce the storage requirements for a table. These compression methodologies can reduce the I/O required to process queries and, when combined with row partitioning, the number of compression opportunities might increase.

The following CREATE TABLE request creates a table definition with four column partitions, two of which are internal partitions that Vantage creates for all column-partitioned tables.

The theoretical maximum number of column partitions for this table is 65,534, including the two column partitions for internal use, and the maximum column partition number is 65,535. The difference is caused by there always being at least one unused column partition number available for altering a column partition.

However, this table can actually have no more than 2,050 column partitions because the total number of columns for a table cannot exceed 2,048. To have 2,050 column partitions, each user-defined column partition could have only one column.

```
CREATE TABLE sales_1 (
  (storeid      INTEGER NOT NULL,          /* Partition 1 */
   productid    INTEGER NOT NULL,
   salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   totalrevenue DECIMAL(13,2)),
  (totalsold    INTEGER,                    /* Partition 2 */
   topsalesperson INTEGER,
   note        VARCHAR(356)))
PRIMARY AMP INDEX (storeid, productid)
PARTITION BY COLUMN; -- Defines 2 multicolumn partitions plus 2
```

```
-- additional column partitions for internal
-- use.
```

The following example creates a table definition for a column-partitioned table with row partitioning.

The COLUMN specification in the PARTITION BY clause defines 7 column partitions: 5 column partitions based on the column grouping in the column definition list plus 2 additional column partitions for internal use.

The maximum number of column partitions is 779 including the 2 column partitions for internal use. This means that it is possible to add 770 column partitions to this table. The maximum column partition number is 780.

The RANGE_N function in the partitioning expression defines 48 row partitions with the option to add up to 36 row partitions for a maximum of 84 row partitions. The maximum row partition number is 84.

The maximum combined partition number for this table is 65,520, which is computed from the following product: 780×84 . This is the product of the maximum partition number of each partitioning level. The table has 2-byte partitioning because the number of combined partitions is not greater than 65,535.

If you were to create this table using multicolumn partitions, you could add columns up to the maximum of 2,048. You could also specify the ADD option to specify a higher number of column partitions that could be added, but that would result in the table using 8-byte partitioning rather than 2-byte partitioning.

```
CREATE TABLE sales_2 (
    storeid      INTEGER NOT NULL,
    productid    INTEGER NOT NULL,
    salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    totalrevenue DECIMAL(13,2),
    (totalsold    INTEGER,
    topsalesperson INTEGER,
    note         VARCHAR(256)))
PRIMARY AMP INDEX (storeid, productid)
PARTITION BY (COLUMN, -- Defines 4 single-column partitions and
                  -- 1 multicolumn partition plus 2
                  -- additional column partitions for internal
                  -- use.
    RANGE_N(salesdate BETWEEN DATE '2007-01-01'
                  AND      DATE '2010-12-31'
                  EACH INTERVAL '1' MONTH)
    ADD 36);
```

You can use column partitioning to improve query performance through column partition elimination.

You can use row partitioning to improve query performance through row partition elimination, which reduces the need to access all of the rows in a table. Vantage uses efficient methods to reconstruct qualifying rows from the projected columns of the column partitions that are not eliminated.

Advantages of column partitioning include a simple CREATE TABLE syntax with default autocompression, the ability of the Optimizer to perform column partition elimination, and to facilitate efficient access data from column partitions. Vantage manages the column partitions so that the table appears to you as a single object.

Column Partitioning Performance

You can exploit column partitioning to improve the performance of some classes of workloads. The general performance impact of column partitioning is summarized in the following bullets.

- You should see a significant I/O performance improvement for requests that access a variable small subset of the columns and rows of a column-partitioned table or join index. This includes accessing columns to respond to both predicates and projections.

For example, if 20% of the data in rows is required to return the result set for a query, the I/O for a column partitioned table should be approximately 20% of the I/O for the same table without column partitioning.

Column partitioning can further reduce I/O depending on the effectiveness of autocompression and row header compression.

Additional I/O might be required to reassemble rows if many columns are projected or specified in query predicates.

- You might see a negative impact on the performance of queries that access more than a small subset of the columns or rows of a column-partitioned table or join index.
- You might see a relative increase in spool size compared to the size of a source column-partitioned table.

When data from an autocompressed column-partitioned table is spooled, Vantage does not carry the autocompression over to the spool because the spool is row-oriented. This can lead to a large spool relative to the compressed data in the column-partitioned table.

Because user-specified compression is carried over to the spool, applying user-specified compression to the column-partitioned table might be beneficial if spool usage in terms of space consumption and I/O operations becomes an issue.

- You might see a reduction in CPU usage for column-partitioned tables.

At the same time, consumption of CPU resources might increase to process autocompression, decompression, and containers.

With a reduction in I/O and a possible increase in CPU, workloads can change from being I/O-bound to being CPU-bound, and the performance of CPU-bound workloads might not improve, and might even be worse, with column partitioning.

- You might see a negative performance impact on INSERT operations for a column-partitioned table or join index, especially for single-row inserts, and less so for block-at-a-time bulk insert operations such as array inserts and INSERT ... SELECT operations.

For an INSERT ... SELECT operation, the CPU cost increases as the number of column partitions increases because there is a cost to split a row into multiple column partitions.

You must understand the potential tradeoffs when you consider the number of column partitions you create for a table. For example, workloads that contain a large number of INSERT operations can benefit from a table with fewer column partitions when it comes to CPU usage, but creating the table with columns in their own individual partitions might be more optimal for space usage and decreasing

the number of I/Os, so you must determine an appropriate balance among the factors that you can finely tune.

For example, a good candidate for column partitioning is a table where the workloads that access it are heavily query-oriented, and the benefits gained from column partitioning, even though the partitioning increases the CPU cost to load the data, a good tradeoff.

Value compression and autocompression can have a negative impact on CPU consumption for workloads that tend to insert many rows, similar to the impact that is seen with a nonpartitioned table that has compression. Because Vantage applies autocompression to every column partition by default, this can cause a significant increase in CPU consumption compared to multivalue compression, which might only be selectively applied to columns.

However, compression can significantly reduce space usage and decrease the I/O operations required for the INSERT operations and for subsequent requests, making the tradeoff between increased CPU consumption and decreased I/O operations a factor that must be considered.

Be aware that FastLoad and MultiLoad are not supported for column-partitioned tables.

- You might see a negative performance impact for UPDATE operations that select more than a small subset of rows to be updated. Because updates are done as a DELETE operation followed by an INSERT operation, Vantage needs to access all of the columns of rows selected for update.

The cost of performing updates that access many rows when the table is column-partitioned might not be acceptable, and if it is not, you should avoid column partitioning the table. For these kinds of UPDATE operations, doing an INSERT ... SELECT request into a copy of the table might be a better alternative.

- Take care not to over-partition tables.

In extreme cases of over-partitioning, a column-partitioned table might be as much as 22 times larger than a table that is not column-partitioned.

Row partitioning of a column-partitioned table might result in over-partitioning such that only a few values with the same combined partition number occur and, so only a few values are included in each of the containers, which reduces the effectiveness of row header compression.

When increasingly more containers are required to respond to a request, each supporting fewer column partition values, the advantage of row header compression is lost. In addition, more I/O is required to access the same amount of useful data. A data block might contain a mix of eliminated and non-eliminated combined partitions for a query. But to read the non-eliminated combined partitions, the entire data block must be read and, therefore, eliminated combined partitions in the data block are also being read unnecessarily.

Over-partitioning may exist when populated combined partitions have fewer than 10 data blocks. With 10 data blocks per combined partition, 10% of the data that Vantage reads is not required by a request. As the number of data blocks decreases, in increasingly large quantity of unneeded data is read. In the worst case, even if there is column partition elimination, all the data blocks of the table must be read.

The magnitude of performance changes when accessing a column-partitioned table or join index, both positive and negative, can range over several orders of magnitude depending on the workload. You should

not column-partition a table when performance is so severely compromised that you cannot offset the reduced performance with physical database design choices such as join indexes.

These and other impacts of column partitioning are described in more detail in the topics that follow.

Cases Where Positive Performance Effects Are Most Likely To Occur

The greatest improvement in the performance of querying column-partitioned tables occurs when a request specifies a highly selective predicate on a column in a single-column partition of a column-partitioned table with hundreds or thousands of columns and only a small number of them are projected by the request.

Cases Where Negative Performance Effects Are Most Likely To Occur

The greatest decrement in the performance of querying column-partitioned tables occurs when the following conditions take place.

- Most or all of the columns in a column-partitioned table are projected by a request.
- The request is not selective.
- The table being queried has thousands of column partitions.
- The retrieval performance for a column-partitioned table or join index is not good when the number of column partition contexts that are available is significantly fewer than the number of column partitions that must be accessed to respond to the query. Note that there are at least eight available column partitions contexts. Depending on your memory configuration, there may be more available contexts.

When this happens, consider reconfiguring the table or join index to decrease the number of column partitions that need to be accessed by combining column partitions so there are fewer of them.

- The table being queried has enough row-partitioned levels that there are very few physical rows in populated combined partitions, and the physical rows contain only one or a few column partition values.

Autocompression

When you create a column-partitioned table or join index, Vantage attempts to use one or more methods to compress the data that you insert into the physical rows of the object unless you specify the NO AUTO COMPRESS option at the time you create it or NO AUTO COMPRESS is set as the default. The process of selecting and applying appropriate compression methods to the physical containers of a column-partitioned table or join index is referred to as *autocompression*.

Autocompression is most effective for a column partition with a single column and COLUMN format.

Vantage only applies autocompression to column partitions with COLUMN format, and then only if it reduces the size of a container. Vantage autocompresses column partitions by default with the following requirements.

- Minimal CPU resources are required to decompress the data for reading.
- Vantage does not need to decompress many values to find a single value.

Vantage applies autocompression for a physical row on a per container basis. For efficiency, the system may use the autocompression method chosen for the previous container, including not using autocompression, if that is more effective. Containers in a column partition might be autocompressed in different ways. In

most cases, the data type of a column is not a factor, and Vantage compresses values based only on their byte representation. As a general rule, the only difference that needs to be considered is whether the byte representation is fixed or variable length.

For some values there are no applicable compression techniques that can reduce the size of the physical row, so Vantage does not compress the values for that physical row, but otherwise the system attempts to compress physical row values using one of the autocompression methods available to it. When you retrieve rows from a column-partitioned table, Vantage automatically decompresses any compressed column partition values as is necessary.

Because the selection of autocompression methods used for a container is made by Vantage and not by users, the methods that autocompression can select from to compress the data in a column partition are not documented in the Teradata user documentation library.

Examples

Consider the following PARTITION BY clause as an example of what happens with excess partitions. Assume that *o_custkey* has the INTEGER data type, *o_orderdate* has the DATE data type, and there are 8 columns defined for the table. Vantage defines 8-byte partitioning because the maximum combined partition number before adding excess partitions to a level is 462,000 $((8+2+1)*500*84)$, which is greater than the maximum combined partition number for 2-byte partitioning, which is 65,335.

```
PARTITION BY (COLUMN,
               RANGE_N(o_custkey BETWEEN 0
                       AND      499999
                       EACH 1000),
               RANGE_N(o_orderdate BETWEEN DATE '2003-01-01'
                       AND      DATE '2009-12-31'
                       EACH INTERVAL '1' MONTH) )
```

The partitioning for this database object has the following characteristics.

- The number of column partitions defined for level 1 is 10, including two internal use column partitions and assuming a single column per partition.

The maximum number of column partitions is 20, meaning that 10 additional column partitions could be added.

The maximum column partition number is 21.

- The number of row partitions defined for level 2 is 500.

Vantage adds any excess partitions to level 2, so level 2 has a maximum of 5,228,668,955,133,092 row partitions.

The default for level 2 is ADD 5228668955132592.

- The number of row partitions defined for level 3 is 84, which is the same number of partitions defined for level 3.

Because this is not the first row partitioning level that does not specify an ADD option, the default is ADD 0.

The implication of all this is that the partitioning specified by the preceding PARTITION BY clause is equivalent to the following PARTITION BY clause.

```
PARTITION BY (COLUMN ADD 10,
              RANGE_N(o_custkey BETWEEN 0
                      AND      499999
                      EACH 1000)
              ADD 5228668955132592,
              RANGE_N(o_orderdate BETWEEN DATE '2003-01-01'
                      AND      DATE '2009-12-31'
                      EACH INTERVAL '1' MONTH)
              ADD 0 )
```

The adjusted maximum combined partition number is 9,223,372,036,854,774,288.

The following example is a full CREATE TABLE request that creates the column-partitioned table named *t1*.

```
CREATE TABLE t1 (
  a01 INTEGER, a02 INTEGER, a03 INTEGER, a04 INTEGER, a05 INTEGER,
  a06 INTEGER, a07 INTEGER, a08 INTEGER, a09 INTEGER, a10 INTEGER,
  a11 INTEGER, a12 INTEGER, a13 INTEGER, a14 INTEGER, a15 INTEGER,
  a16 INTEGER, a17 INTEGER, a18 INTEGER, a19 INTEGER, a20 INTEGER,
  a21 INTEGER, a22 INTEGER, a23 INTEGER, a24 INTEGER, a25 INTEGER,
  a26 INTEGER, a27 INTEGER, a28 INTEGER, a29 INTEGER, a30 INTEGER,
  a31 INTEGER, a32 INTEGER, a33 INTEGER, a34 INTEGER, a35 INTEGER,
  a36 INTEGER, a37 INTEGER, a38 INTEGER, a39 INTEGER, a40 INTEGER,
  a41 INTEGER, a42 INTEGER, a43 INTEGER, a44 INTEGER, a45 INTEGER,
  a46 INTEGER, a47 INTEGER, a48 INTEGER, a49 INTEGER, a50 INTEGER,
  a51 INTEGER, a52 INTEGER, a53 INTEGER, a54 INTEGER, a55 INTEGER,
  a56 INTEGER, a57 INTEGER, a58 INTEGER, a59 INTEGER, a60 INTEGER,
  a61 INTEGER, a62 INTEGER, a63 INTEGER, a64 INTEGER, a65 INTEGER,
  a66 INTEGER, a67 INTEGER, a68 INTEGER, a69 INTEGER, a70 INTEGER,
  a71 INTEGER, a72 INTEGER, a73 INTEGER, a74 INTEGER, a75 INTEGER,
  a76 INTEGER, a77 INTEGER, a78 INTEGER, a79 INTEGER, a80 INTEGER,
  a81 INTEGER, a82 INTEGER, a83 INTEGER, a84 INTEGER, a85 INTEGER,
  a86 INTEGER, a87 INTEGER, a88 INTEGER, a89 INTEGER, a90 INTEGER,
  a91 INTEGER, a92 INTEGER, a93 INTEGER, a94 INTEGER, a95 INTEGER,
  a96 INTEGER, a97 INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a2 BETWEEN 1
                      AND      48
                      EACH 1,
```



```

NO RANGE, UNKNOWN)
ADD 10,
COLUMN,
RANGE_N(a3 BETWEEN 1
        AND 50
        EACH 10));

```

The partitioning for this table has the following characteristics.

- The number of partitions defined for level 1 is 50 and, initially, the maximum number of partitions and the maximum partition number for this level is $(50+10) = 60$ because there is an ADD 10 clause for this level.
- The number of partitions defined for level 2 is 99: 97 user-specified partitions plus 2 for internal use.

Because there is a level of row partitioning without an ADD clause, the column partitioning level has a default of ADD 10, so the maximum number of partitions for this level is 109, with a maximum column partition number of 110.

- The number of partitions defined for level 3 is 5. Because there is no ADD clause specified for this level and it is the first row partitioning level without an ADD clause, assuming a default of ADD 0, the maximum combined partition number before adding excess partitions to a level is $(60*110*5)$, or 33,000, which is not greater than 65,535, so this partitioning consumes 2 bytes in the row header.

The actual maximum number of partitions for this level is the largest number that would not cause the maximum combined partition number to exceed 65,535.

This level has a default of ADD 4, and the maximum number of partitions for the level is 9.

The adjusted maximum combined partition number for the table is $(60*110*9)$, or 59,400.

Any remaining excess partitions can be added to level 1, meaning that the maximum number of partitions for level 1 can be increased such that the maximum combined partition number does not exceed 65,535.

The ADD 10 clause for level 1 can be replaced by ADD 16.

The implication of all this is that the partitioning specified by the preceding CREATE TABLE request is equivalent to the following CREATE TABLE request.

```

CREATE TABLE t1 (
  a01 INTEGER, a02 INTEGER, a03 INTEGER, a04 INTEGER, a05 INTEGER,
  a06 INTEGER, a07 INTEGER, a08 INTEGER, a09 INTEGER, a10 INTEGER,
  a11 INTEGER, a12 INTEGER, a13 INTEGER, a14 INTEGER, a15 INTEGER,
  a16 INTEGER, a17 INTEGER, a18 INTEGER, a19 INTEGER, a20 INTEGER,
  a21 INTEGER, a22 INTEGER, a23 INTEGER, a24 INTEGER, a25 INTEGER,
  a26 INTEGER, a27 INTEGER, a28 INTEGER, a29 INTEGER, a30 INTEGER,
  a31 INTEGER, a32 INTEGER, a33 INTEGER, a34 INTEGER, a35 INTEGER,
  a36 INTEGER, a37 INTEGER, a38 INTEGER, a39 INTEGER, a40 INTEGER,
  a41 INTEGER, a42 INTEGER, a43 INTEGER, a44 INTEGER, a45 INTEGER,
  a46 INTEGER, a47 INTEGER, a48 INTEGER, a49 INTEGER, a50 INTEGER,
  a51 INTEGER, a52 INTEGER, a53 INTEGER, a54 INTEGER, a55 INTEGER,

```

```

a56 INTEGER, a57 INTEGER, a58 INTEGER, a59 INTEGER, a60 INTEGER,
a61 INTEGER, a62 INTEGER, a63 INTEGER, a64 INTEGER, a65 INTEGER,
a66 INTEGER, a67 INTEGER, a68 INTEGER, a69 INTEGER, a70 INTEGER,
a71 INTEGER, a72 INTEGER, a73 INTEGER, a74 INTEGER, a75 INTEGER,
a76 INTEGER, a77 INTEGER, a78 INTEGER, a79 INTEGER, a80 INTEGER,
a81 INTEGER, a82 INTEGER, a83 INTEGER, a84 INTEGER, a85 INTEGER,
a86 INTEGER, a87 INTEGER, a88 INTEGER, a89 INTEGER, a90 INTEGER,
a91 INTEGER, a92 INTEGER, a93 INTEGER, a94 INTEGER, a95 INTEGER,
a96 INTEGER, a97 INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a2 BETWEEN 1
                    AND    48
                    EACH 1,
NO RANGE, UNKNOWN)
ADD 16,
COLUMN ADD 10,
RANGE_N(a3 BETWEEN 1
                    AND    50
                    EACH 10)
ADD 4);

```

Now the maximum combined partition number is $(66 \times 110 \times 9)$, or 65,340. The maximum number of combined partitions is $(66 \times 109 \times 9)$, or 64,746. The number of combined partitions is $(50 \times 99 \times 5)$, or 24,750 and you can alter the table to add additional partitions to each of the partitioning levels.

Autocompression and Spools

A spool generated from a table with autocompression does not have any autocompression, though it might have inherited user-specified compression based on the rules for spool inheriting the compression characteristics of its parent. This means a very large spool relative to the size of the column partitions can be generated if those column partitions are highly compressed by the autocompression and there is little or no selectivity.

Autocompression Interactions With User-Specified Compression Methods

Vantage applies user-specified compression for columns within a multicolumn column partition value, but Vantage applies autocompression to a column partition value as a whole.

Vantage might decide not to use one or more of the user-specified compression techniques with autocompression if it determines that other autocompression techniques, including none, provide better compression for a container.

When the system constructs a container, it first applies any user-specified compression. When the container reaches a certain size limit, which is defined internally, Vantage examines the container to determine what autocompression techniques and user-specified compression can be used to reduce its size. After compressing the container, the system can append additional column partition values to it using the method

of autocompression it has determined to be optimal and any user-specified techniques until a container size limit is reached, at which time Vantage constructs a new container.

As an alternative to the previous method, the system may decide that it is more efficient to reuse the compression from the previous container and apply that compression when constructing a container.

If you specify block-level compression, which is not an autocompression technique, for a column-partitioned table or join index, it is applied for data blocks independently of whether Vantage applies autocompression.

Checking the Effectiveness of Autocompression

To check the effectiveness of autocompression, use the SHOWBLOCKS command of the Ferret utility to compare the size of the data or a sample of the data stored with and without compression. You should also compare the performance with and without compression to measure the benefits of each.

SHOWBLOCKS is documented in *Utilities*.

Using the NO AUTO COMPRESS Option

You can override the autocompression default by specifying the NO AUTO COMPRESS option.

If you specify NO AUTO COMPRESS for a column partition, or if the column partition has ROW format, Vantage always applies any compression techniques (null compression, multivalued compression, or algorithmic compression) that you specify to every container or subrow of the column partition. For column partitions with COLUMN format, Vantage applies row header compression, which is applied to column partitions that have COLUMN format. To disable row header compression, specify ROW format when you create a column-partitioned table or join index.

There is some overhead in determining whether or not a physical row should be compressed and, if so, what compression techniques to use. If Vantage determines that there is no appropriate technique to compress the physical rows of the column partition, or if you determine that the compression techniques used do not effectively compress the column partition, you can eliminate this overhead by specifying the NO AUTO COMPRESS option for the column partition.

Anticipated Workload Characteristics for Column-Partitioned Tables and Join Indexes

The expected scenarios for column-partitioned tables and join indexes are those where the partitioned table data is loaded with an INSERT ... SELECT request with possibly some minor ongoing maintenance, and then the table is used to run analytics and data mining, at which point the table or row partitions are deleted, and the scenario then begins over again. This is referred to as an insert once scenario. Column-partitioned tables are not intended to be used for OLTP- or tactical query-type activities.

The design point for data maintenance of column-partitioned database objects is roughly 88% INSERT ... SELECT or array INSERT operations into empty table or row partitions, 2% other inserts operations, 7% update operations, and 3% delete operations.

Most requests that access a column-partitioned table or join index are expected to be selective on a variable subset of columns, or to project a variable subset of the columns, where the subset accesses fewer than 10% of the column partitions for any particular request.

The expected number of column partitions that need to be accessed for requests should preferably not exceed the number of available column partition contexts as it does in the following EXPLAIN of a SELECT request. If it does, there may be undesirable negative impact on performance in some cases.

Note that the count of 21 for the number of column partitions includes the 20 selected partitions in the SELECT statement and the delete column partition from the column-partitioned table. 21 exceeds the number of available column partition contexts, so 20 column partitions (including the delete column partition) of the column-partitioned table are merged into the first subrow column partition of the column-partitioned merge spool. The remaining column partition that needs to be accessed from the column-partitioned table is copied to the second subrow column partition in the column-partitioned merge spool (for a total of 2 subrow column partitions). This reduces the number of column partitions to be accessed at one time (which is limited by the number of available column-partition contexts). The result is then retrieved from the two subrow column partitions of the column-partitioned merge spool.

```
EXPLAIN SELECT a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, r,
              s, t, u
FROM /*CP*/ t1;
```

```
*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.
```

Explanation

-
- 1) First, we lock PLS.t1 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we do an all-AMPs RETRIEVE step from 21 column partitions (20 contexts) of PLS.t1 using covering CP merge Spool 2 (2 subrow partitions and Last Use) by way of an all-rows scan with no residual conditions into Spool 1 (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 2 rows (614 bytes). The estimated time for this step is 0.03 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
 - > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

A column-partitioned table can be used as a sandbox table where data can be added until an appropriate indexing method is determined. Requests that access a small, but variable, subset of the columns might run more efficiently against a column-partitioned table compared to a table without column partitioning.

General Performance Guidelines for Column Partitioning

Use these guidelines as a starting point, but understand that they might not be suitable for all workloads. As you gain experience using column partitioning, you might find that alternate choices are more appropriate in some cases.

Note the following general points about optimizing the performance of column-partitioned tables and join indexes.

- Keep in mind that column partitioning is not optimal for all query types and workloads.
- As is always true, you should test and prove any physical database design, preferably first on a test system with a valid sample of the data and then on the production system with the full data before releasing the design into the production environment.

This includes testing all of the following items before putting a column-partitioned table or join index into production use.

- Queries
- Workloads
- Bulk data loads
- Maintenance
- Archive, restore, and copy performance

Also be sure to check the space usage of the tables.

The performance guidelines for column-partitioned tables and join indexes are divided into the following subgroups.

- General performance guidelines for column partitioning
- Guidelines for queries, contexts, and table maintenance
- Guidelines for partitioning column-partitioned tables and join indexes
- Guidelines for specifying table and column attributes for column-partitioned tables
- Guidelines for specifying compression for column-partitioned tables and join indexes
- Guidelines on I/O operations, CPU usage, and disk space usage for column-partitioned tables and join indexes
- Guidelines for collecting statistics on column-partitioned tables and join indexes

Guidelines for Queries, Contexts, and Table Maintenance for Column Partitioning

- The optimal application for column-partitioned tables is large fact tables or call detail tables that are accessed frequently by analytic workloads.

A join index might provide acceptable performance for requests that depend on primary index access, while column partitioning the base table could enable superior performance for other classes of workloads.

Use column partitioning for tables accessed by analytic queries where the tables are refreshed using bulk loading techniques periodically with possibly some interim modifications.

Do not use column-partitioned tables for highly volatile data, for cases in which a table is lightly populated with rows, or if the data is to be used in tactical query workloads.

- To facilitate query performance, you should make sure that you follow at least one, if not both, of the following guidelines.

- Ensure that the number of column partitions accessed by a request does not exceed the number of available column partition contexts.
- Write the request in such a way that it is highly selective.

Ideally, you should ensure that both of these guidelines are followed.

Requests in workloads that heavily access column-partitioned tables and do not conform to this recommendation should be minimized, because their performance is likely to be degraded.

To support acceptable performance for queries that requests that have the desired characteristics, you should employ physical database design options like secondary and join indexes when possible. Before you employ indexes, you must understand that additional maintenance costs are always incurred because Vantage must update index subtables any time the base table columns they are defined on are updated.

- Always measure the INSERT cost for tables that are candidates for column partitioning.

Do not use column partitioning when the increased cost of inserting data is not acceptable or is not offset by improved performance in the query workload.

- Because you should not use column-partitioned tables for highly volatile data, apply UPDATE operations to column-partitioned tables sparingly.
- Perform DELETE operations on a column-partitioned table either for the entire table or for entire row partitions.
- The Optimizer uses the DBS Control field PPICacheThrP to determine the number of available file contexts that can be used at one time to access a partitioned table.

If the number of available file contexts determined by PPICacheThrP is fewer than 8, then 8 file contexts are available if the maximum block size for the table is 128 MB or less. If the maximum block size is greater than 128 MB, the minimum number of contexts may be reduced to less than 8 but will be at least 2. If the number of file contexts specified by PPICacheThrP is more than 256, then 256 file contexts are available. (This is if the maximum block size for the table is 128K. If the maximum block size is larger, the number of available file contexts may be smaller.) For a column-partitioned database object, Vantage uses the number of file contexts as the number of available column partition contexts.

A file context can be associated with each column partition context for some operations, but in other cases, Vantage might allocate a buffer to be associated with each column partition context.

The ideal number of column partition contexts should be at least equal to the number of column partitions that need to be accessed by a query; otherwise, performance can degrade since not all the needed column partitions can be read at one time. Performance and memory usage can be impacted if PPICacheThrP is set too high, because too high a setting can lead to memory thrashing or even a system crash.

At the same time, the benefits of partitioning can be lessened if PPICacheThrP is set unnecessarily low, and performance might degrade significantly. The default setting is expected to be optimal for most workloads (with the standard 80 AMP worker tasks per AMP); however, after monitoring performance and memory usage, you might need to adjust the setting to obtain the best balance.

- You should periodically refresh or append new rows to a column-partitioned table, or to the row partitions of the column-partitioned table, using INSERT ... SELECT requests that move large quantities of data.

Date or timestamp partitioning may help to improve column-partitioned table maintenance.

Guidelines for Partitioning Column-Partitioned Tables and Join Indexes

- While you can define column partitioning at any level of multilevel partitioning, in most cases you should follow these guidelines when you configure the partitioning for a column-partitioned table.
 - Code the column partitioning level first and follow the column partitioning level with any row partitioning levels.
 - If you do not code the column partition at the first level, code it as the second level after DATE or TIMESTAMP row partitioning.

Some considerations that might lead to putting the column partitioning at a lower level are the following.

- Potential improvements for cylinder migration.
- Block compression effectiveness.
- If you specify row partitioning as part of a multilevel column partitioning for a table, consider specifying the ADD option for the any partitioning levels that might need to increase their number of partitions in the future.
- Unless you have a good reason not to, you should use the defaults when you specify PARTITION BY COLUMN. Do not override the defaults without first giving the reasons for doing so serious consideration
- For columns that are often specified in queries, but where the specific set of columns specified varies from request to request, you should create single-column partitions for the frequently specified columns.
- Use ROW format for wide column partitions because it has less overhead than a container that holds one or a few values.

If Vantage does not assign ROW format for a column partition, but you have determined that ROW format is more appropriate because it decreases space usage, specify ROW explicitly.

- Use COLUMN format for narrow column partitions, especially if autocompression is effective.

If Vantage does not assign COLUMN format for a multicolumn partition, but COLUMN is user-determined to be more appropriate (decreases space usage, etc.), specify COLUMN explicitly.

You might need to specify COLUMN format explicitly for a multicolumn partition that contains a column with a VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC, or VARBYTE data type and defined with a large maximum value, but where values are actually very short in most cases. This is because the system-determined format might be ROW based on the large maximum length.

Guidelines for Specifying Table and Column Attributes for Column Partitioning

- Both NoPI tables and column-partitioned tables are created with a MULTiset table type by default.

Neither NoPI tables nor column-partitioned tables can be created as a SET table. As a result, Vantage does not perform duplicate row checks for either.

- Use the table-level option `DATABLOCKSIZE` if the default data block size is appropriate for the table. Generally, a data block size of 127.5 KB or less is recommended for a column-partitioned table.
- The settings for the table-level option `FREESPACE` and the DBS Control field `FreeSpacePercent` might require adjustment for a column-partitioned table or join index with small internal partitions, as might be the case if a table or join index is also row partitioned, particularly if you add data incrementally to the table or index.

These options specify the amount of space on each cylinder that is to be left unused during load operations. Reserved free space allows tables to expand within their currently allocated cylinders. This can prevent or delay the need for additional cylinders to be allocated, which incurs the overhead of moving data to the new cylinders. Avoiding new cylinder allocations can improve overall system performance.

If you do a large `INSERT ... SELECT` operation and internal partitions are either large or empty, little or no free space is needed. Keeping new table data physically close to existing table data and avoiding data migrations, can improve overall system performance.

- Follow these column attribute guidelines when you create your column-partitioned tables.
 - Specify `NOT NULL` for columns that should not be null.
Because nullable columns can significantly decrease the effectiveness of autocompression, you should avoid them unless you have a sound reason for specifying them.
 - Specify column-level `CHECK` constraints when you can.
`CHECK` constraints are valid for both NoPI tables and column-partitioned tables.
 - Specify `UNIQUE` and `PRIMARY KEY` constraints when you can.
`UNIQUE` and `PRIMARY KEY` constraints are valid for both NoPI tables and column-partitioned tables.
 - Specify foreign key constraints whenever they are applicable.
Foreign key constraints are valid for both NoPI tables and column-partitioned tables.
- The following features are valid for both NoPI tables and for column-partitioned tables:
 - Fallback
 - Unique secondary indexes
 - Nonunique secondary indexes
 - Join indexes
 - Reference indexes
- You cannot specify permanent journaling for a column-partitioned table or a NoPI table.
- The following features are not shared by nonpartitioned NoPI tables and column-partitioned tables.
 - You cannot create a column-partitioned global temporary or volatile table.
A nonpartitioned NoPI table can be created as a global temporary or volatile table.

- A column-partitioned table can have an identity column, while a nonpartitioned NoPI table cannot.
- The setting of the DBS Control field PrimaryIndexDefault does not affect the default primary index specification if PARTITION BY is specified. The default behavior for creating a partitioned table without specifying a primary index or primary AMP index is always NO PRIMARY INDEX.
- You are not required to specify NO PRIMARY INDEX when you create a column-partitioned table, but you may need to specify NO PRIMARY INDEX for a nonpartitioned NoPI table, depending on the DBS Control settings.

The default behavior for CREATE TABLE for a column-partitioned table is NO PRIMARY INDEX if you do not specify a primary index. The setting of the DBS Control field PrimaryIndexDefault does not affect this behavior.

- XML, BLOB, and CLOB are valid data types for both nonpartitioned NoPI tables and column-partitioned tables.

Note:

There is a limit of 256M rows per rowkey per AMP with XML, BLOB, and CLOB data types. NoPI tables normally have only one hash value on each AMP, so the effective limit on the number of rows per AMP is approximately 256M.

The exact number is 268,435,455 rows per rowkey per AMP.

For column-partitioned tables, these limits are per column partition:hash bucket combination rather than PA and NoPI rows per hash value.

Guidelines for Specifying Compression for Column-Partitioned Tables and Join Indexes

- If autocompression is effective for a column, put that column into a single-column partition even if it is not frequently accessed.
- If autocompression on the individual columns or subsets of columns is not effective, you should group columns into a column partition when those columns are always or frequently accessed together by queries or are infrequently accessed.
- If autocompression is not effective for a column partition, specify NO AUTO COMPRESS for the column partition (with COLUMN format) to avoid the overhead of checking for autocompression opportunities when there are none to exploit.

Autocompression is most effective for single-column partitions with COLUMN format, less so for multicolumn partitions (especially as the number of columns increases) with COLUMN format, but not effective for column partitions with ROW format.

- Specify multivalue compression, algorithmic compression, or both for known high-occurrence values for columns where compression can be effective. For example, if you know that the data for a column is limited to a few values, you should specify multivalue compression for those values.

Specify an appropriate algorithmic compression for Unicode columns that contain a substantial number of compressible characters. The ASCII script (U+0000 to U+007F) of Unicode is compressible with the UTF8 algorithm.

Guidelines on Optimizing I/O Operations, CPU Usage, and Disk Space Usage for Column-Partitioned Tables and Join Indexes

- The primary intent of column partitioning is to reduce the number of I/O operations undertaken by a query workload. The following factors can all contribute to reducing the I/O required by query workloads on column-partitioned tables.
 - Column partition elimination
 - Row partition elimination for multilevel partitioned tables and join indexes
 - Highly selective query predicates

Other factors such as those from the following list can also play a role in reducing the number of I/O operations required to resolves a query.

- Autocompression
- Row header compression
- User-specified multivalue and algorithmic compression

Trading I/O for CPU might enhance the performance of many workloads on an I/O-bound system.

- A secondary intent of column partitioning is to reduce the amount of disk space consumed by table and join index storage. This is particularly effective when Vantage can apply row header compression and autocompression to column-partitioned table and join index data.
- Although column partitioning is designed to reduce the number of I/O operations required to process workloads, it is not intended to reduce the CPU usage of queries on column-partitioned tables.

While there are cases where CPU usage decreases for queries made on a column-partitioned table, CPU usage can also increase for some functions such as INSERT operations undertaken on a column-partitioned table.

For a CPU bound system, column partitioning might not provide any benefit, and might even degrade performance. An exception is the case where a subset of the workload that is I/O bound, even if overall the system is CPU bound, in which case column partitioning could be beneficial. Experiment with running your CPU-bound workloads against both nonpartitioned tables and column-partitioned tables to determine what the differences are.

Guidelines for Collecting Statistics on Column-Partitioned Tables and Join Indexes

- Collect statistics regularly on the columns and indexes of a column-partitioned table just as you would for any other tables.
- Always collect statistics on the system-derived PARTITION column.

Deleting Rows From a Column-Partitioned Table

Consider the following information before deleting data from a column-partitioned table.

- A DELETE ALL request or an unconstrained DELETE request takes the fastpath DELETE for any table. If specified within an explicit transaction in Teradata session mode, the DELETE request must be the last statement of the last request of the transaction.

Similarly, if the column-partitioned table is also row-partitioned, Vantage can do a fastpath DELETE. For these cases, Vantage recovers the space that had been used by the deleted rows.

A *fastpath* optimization is one that can be performed faster if certain conditions are met. For example, in some circumstances DELETE and INSERT operations can be performed faster if they can avoid reading the data blocks and avoid transient journaling.

- For all other cases, a DELETE request uses a scan or an index on a column-partitioned table. In this case, the rows are marked as deleted in the delete column partition without recovering the space, but both LOB space and index space is recovered. If column partitions with ROW format do have their space deleted and if all column partitions have ROW format, the row is not marked as deleted in the delete column partition.

Because of this, you should only delete rows in this manner for a relatively small number of rows and you should use the form of DELETE request described in the previous bullet to delete large amounts of rows.

The space is recovered from the column-partitioned table when all the rows are deleted at the end of a transaction or when the entire row partition that contains the deleted rows is deleted at the end of a transaction.

Updating a Column-Partitioned Table

Consider the following information before updating data in a column-partitioned table.

- An UPDATE request uses a scan, an index, or a rowID spool to access a column-partitioned table and select the qualifying rows for the update.
- An UPDATE request is processed in the following way.
 1. Selects the rows to be updated.
 2. Transforms columns to rows.
 3. Deletes the old row without recovering the space and marks its delete bit in the delete column partition.

Note:

Both LOB space and index space is recovered.

4. Reinserts the updated rows, transforming them from rows to columns and appending the column values to their corresponding combined partitions.

Vantage recovers the space from the column-partitioned table when it deletes all of the rows at the end of a transaction or when it deletes the entire row partition that contains the deleted rows at the end of a transaction.

Note:

If the columns being updated are only in column partitions of a table with ROW format (and the columns being updated are not primary AMP index, primary index, or partitioning columns), the update is made in place instead of as a delete of the old row and an insert of the new row.

- Vantage also updates the columns in the column-partitioned table that are used in a secondary or join index.

Operations and Utilities for Column-Partitioned Tables

The following tables provide a list of the operations and utilities that you might consider using with column-partitioned tables. The tables include information such as whether the utility or operation is supported for column-partitioned tables, possible substitutions for utilities and operations that Vantage does not support for column-partitioned tables, and various usage information about the operation or utility as it applies to nonpartitioned NoPI tables.

The first table lists the SQL operations you might use for column-partitioned tables.

SQL Statement Name	Support for Column-Partitioned Tables	Usage Notes
DELETE	Supported	See Deleting Rows From a Column-Partitioned Table .
INSERT	Supported	While INSERT operations into column-partitioned tables are supported, you should not use single-row INSERT requests to add rows to column-partitioned tables frequently because such requests can cause a large degradation in performance by needing to transform a row into a column and then appending a column partition value to each of the column partitions.
INSERT ... SELECT	Supported	
MERGE	Not supported	The UPDATE component of a MERGE request is required to fully specify the primary index. Vantage does not support MERGE requests for NoPI and column-partitioned tables. Use UPDATE and INSERT requests instead.
UPDATE	Supported	See Updating a Column-Partitioned Table .
UPDATE (Upsert Form)	Not supported	The UPDATE component of an Upsert request is required to fully specify the primary index. Vantage does not support UPSERT requests for NoPI and column-partitioned tables.

The second table lists the Teradata Tools and Utilities operations you might use for column-partitioned tables.

TTU Utility Name	Support for Column-Partitioned Tables	Usage Notes
CheckTable	Supported	The LEVEL HASH check, which is done with either an explicit LEVEL HASH command or implicitly in a LEVEL THREE command,

TTU Utility Name	Support for Column-Partitioned Tables	Usage Notes
		<p>works differently on a primary-indexed table and a NoPI or column-partitioned table.</p> <ul style="list-style-type: none"> For a primary-indexed table, the check regenerates the row hash for each data row based on the primary index values and then compares with the rows on disk. For NoPI tables and column-partitioned tables, the check looks at the row hash value for each data row and verifies that the hash bucket that is part of the row hash correctly belongs to the AMP.
FastExport	Supported	You can export the data in NoPI tables and column-partitioned tables the same way you can export data rows for a primary-indexed table
FastLoad	Not supported	<p>Use INSERT ... SELECT or Teradata Parallel Data Pump requests instead. These are normally used to populate a column-partitioned table.</p> <p>You can also use FastLoad to load data into a staging table and then use an INSERT ... SELECT request to populate the column-partitioned table</p>
MultiLoad	Not supported	Use INSERT ... SELECT and Teradata Parallel Data Pump requests instead. These are normally used to populate a column-partitioned table.
Reconfiguration	Supported	<p>Reconfiguration processing for a NoPI table or column-partitioned table is similar to the Reconfiguration processing that is done for a primary-indexed table.</p> <p>The main difference is that a NoPI table or column-partitioned table normally has one hash bucket per AMP, which is like a very skewed NUPI table.</p> <p>Because of this, when you reconfigure a system to have more AMPs, there might be some AMPs that do not have any data for a NoPI table or column-partitioned table.</p> <p>As a result, Reconfiguration can cause data skewing for both NoPI and column-partitioned tables.</p>
Restore and Copy	Supported	<p>Restore and Copy processing for a NoPI or column-partitioned table are very similar to the processing used by those utilities for a primary-indexed table.</p> <p>The main difference is that a NoPI table or column-partitioned table normally has one hash bucket per AMP, which is like a very nonunique NUPI table.</p> <p>Because of this, when you restore or copy the data from a NoPI table to a different configuration that has more AMPs, there might be some AMPs that do not have any data.</p> <p>As a result, Restore and Copy can cause data skewing for both NoPI and column-partitioned tables.</p>
TableRebuild	Supported	Table Rebuild processing for a NoPI or column-partitioned table is the same as the Table Rebuild processing for a primary-indexed table.

TTU Utility Name	Support for Column-Partitioned Tables	Usage Notes
		The table must have fallback protection for Table Rebuild to be able to rebuild it. Rows in a NoPI table or column-partitioned table have a rowid, so Vantage can rebuild them the same way it rebuilds rows for a primary-indexed table.

Storage and Other Overhead Considerations for Partitioning

Column Partitions With COLUMN Format

A column partition with COLUMN format packs column partition values into a physical row, or container, up to a system-determined limit. The column partition values must be in the same combined partition to be packed into a container.

The row header occurs once for a container instead of there being a row header for each column partition value.

The format for the row header is either 14 or 20 bytes and consists of the following fields.

- Length
- rowID

The rowID of the first column partition value is the rowID of the container.

- Flag byte
- First presence byte

The rowID of a column partition value can be determined by its position within the container. If many column partition values can be packed into a container, this row header compression can greatly reduce the space needed for a column-partitioned object compared to the same object without column partitioning.

If Vantage can only place a few column partition values in a container because of their width, there can actually be an increase in the space needed for a column-partitioned object compared to the object without column partitioning. In this case, ROW format might be more appropriate.

If Vantage can only place a few column partition values because the row partitioning is such that only a few column partition values occur for each combined partition, there can be a very large increase in the space needed for a column-partitioned object compared to the object without column partitioning. In the worst case, the space required for the column-partitioned object can be as much as 24 times larger.

In this case, consider altering the row partitioning to allow for more column partition values per combined partition or removing column partitioning.

If the container has autocompression, 2 bytes are used as an offset to compression bits, 1 or more bytes indicate the autocompression types and their arguments, if any, for the container, 1 or bytes, depending

on the number of column partition values, of autocompression bits, 0 or more bytes are used for a local value-list dictionary depending the autocompression type, and 0 or more bytes are used for present column partition values.

If the container does not have autocompression either because you specified NO AUTO COMPRESS for the column partition or because no autocompression types are applicable for the column partition values of the container, Vantage uses 0 or more bytes for column partition values.

The byte length of a container is rounded up to a multiple of 8.

The formats for single-column and multicolumn partitions with COLUMN format differ slightly, as listed by the following bullets.

- A single-column partition with COLUMN format consists only of single-column containers. Each container represents a series of column values.
- A multicolumn partition with COLUMN format consists of multicolumn containers. Each container represents a series of column partition values where the column partition value is made up of a set of values, one for each column in the partition.

See [Row Structure for Containers \(COLUMN Format\)](#) for more information.

Column Partitions With ROW Format

A physical row of a column partition with ROW format is called a subrow. Subrows have the same format as traditional database rows, but only include the columns defined for the column partition. Note that currently there is no autocompression for subrows.

If Vantage uses ROW format for the column partitions of a column-partitioned object is such that many narrow column partition values occur, there can be a very large increase in the space needed for a column-partitioned object compared to the object without column partitioning. In the worst case, the space required for the column-partitioned object can be as much as 24 times larger.

Because of this, you should not specify ROW format for narrow column partitions. In the worst case, each column partition value can have a row header (14 or 20 bytes plus 6 or more bytes needed for a container) for each column partition value as compared to the same object without column partitioning. An object that is not column-partitioned only has a row header (14 or 20 bytes) for each regular row.

ROW format is useful for wide column partitions where one or only a few values fit in a container, and there is neither much benefit nor a negative impact from row header compression. ROW format provides quicker and more direct access to a specific column partition value than with COLUMN format because when a container has COLUMN format, Vantage must locate the column partition value within the container.

Depending on the autocompression types used for a container, access can be as simple as indexing into the container or it might require a sequential access through bits indicating how a value is compressed or sequential access through the column partition values, or both, to position to the specific column partition value to be accessed.

See [Row Structure for Subrows \(ROW Format\)](#) for more information.

Advantages and Disadvantages of Partitioned Primary Indexes

Advantages	Disadvantages
<ul style="list-style-type: none"> Row partition elimination enables large performance gains to be realizable, and these are visible to end users. For this particular optimization, more populated partitions are generally better than fewer populated partitions. Batch inserts and updates can run faster if the partitioning schema matches the data arrival pattern. This optimization is visible only to the DBA and operations staff. <ul style="list-style-type: none"> Finer granularity of partitions is generally better than a coarser granularity: daily is better than monthly. The largest performance improvements occur when there are no secondary indexes. Teradata Parallel Data Pump inserts and updates can benefit from more FSG cache hits because of the increased locality of reference when a target table is partitioned on transaction date. In this case, a finer partition granularity is generally better than a coarser partition granularity. Inserts into empty partitions are not journaled. This optimization is only invoked if the table has no referential integrity constraints. 	<ul style="list-style-type: none"> Partitioned table rows are each 2 or 8 bytes wider than the equivalent nonpartitioned table row. Partitioned table rows are 4 bytes wider if multivalue compression is specified for the table. The extra 2 or 8 bytes are used to store the internal partition number for the row. You cannot define the primary index of a partitioned table to be unique unless the entire partitioning column set is part of the primary index definition. You can define a USI on the primary index columns to enforce uniqueness if the complete partitioning column set is not a component of the primary index definition; however, that adds different performance issues. Primary index-based row access can be degraded if the partitioning column set is not a component of the primary index. <ul style="list-style-type: none"> If you can define a secondary index on the primary index column set, then performance is independent of the number of partitions. If you cannot, or have not, defined a secondary index on the primary index, then having fewer partitions is better than having more partitions, whether achieved by means of the table definition itself or by row partition elimination during query processing.
<ul style="list-style-type: none"> Delete operations can be nearly instantaneous when the partitioning column set matches the retention policy, there is no secondary index defined on the partitioning column set, and the delete is the last statement in the transaction. You can delete all of the rows in a partition if you want to do so. In this case, there is no journaling of rows if no secondary index is defined on the partitioning column set. These properties might permit you to drop a secondary index or join index on the partitioning column set. 	<ul style="list-style-type: none"> Joins of partitioned tables to nonpartitioned tables with the same primary index can be degraded. To combat this, observe the following guidelines: <ul style="list-style-type: none"> Identically partition all tables to be joined with the same primary index when possible and then join them on the partitioning columns. Fewer partitions, whether achieved by means of the table definition itself or by row partition elimination, are often better than more partitions for the nonpartitioned-to-partitioned join scenario.

Usage Recommendations For Row Partitioning

Row Partition Elimination

Row partition elimination is a method for enhancing query performance against row-partitioned tables by skipping row partitions that do not contain rows that meet the search conditions of a query. Row partition

elimination is an automatic optimization in which the Optimizer (or, in the case of dynamic row partition elimination, the AMP software) determines, based on query conditions and a row partitioning expression, that some partitions for that partitioning expression cannot contain qualifying rows; therefore, those row partitions can be skipped during a file scan.

Note:

The Optimizer cannot exert all optimizations that are possible for each of the individual types of row partition elimination.

Partitions that are skipped for a particular query are called *eliminated* row partitions.

When there are multiple partitioning expressions, Vantage combines row partition elimination at each of the levels to further reduce the number of data subsets that need to be scanned. For most applications, the greatest benefit of row partitioning is obtained from row partition elimination.

Basing Partitioning on Modulo Partitioning of a Numeric Column

This form uses a single column with a wide range of values, and maximizes the number of partitions. As an example, assume that a telephone company has a table with detailed information about each outgoing telephone call. One of the columns is the originating phone number, defined with the DECIMAL data type. A possible, and perhaps useful, partitioning expression can be defined as follows:

```
CREATE TABLE
...
PRIMARY INDEX (phone_number, call_start)
PARTITION BY phone_number mod 65535 + 1;
```

This partitioning expression, assuming millions of subscribers, populates each of the 65,535 partitions. Some partitions might have more rows than others, because some customers make more phone calls than others, but the distribution among the partitions should be somewhat even. This partitioning expression can improve the performance of a query that examines all phone calls made from a particular number by orders of magnitude by scanning only one partition out of 65,535 instead of the entire table.

Some disadvantages of this form are that the partitioning cannot be altered unless the table is empty, a maximum on 65535 partitions can be defined, and row partition elimination for queries is usually limited to constant or USING value equality conditions on the partitioning column.

Basing the Partitioning Expression on Two or More Numeric Columns

This form uses arithmetic operations, typically multiplication and addition, on two or more numeric columns with suitably small value ranges. Assume a table with a three-digit product code and a two-digit store number. The store numbers count consecutively from 0, and there are fewer than 65 stores. This table can be partitioned as follows:

```
CREATE TABLE
...
```

```
PRIMARY INDEX (store_number, product_code, sales_date)
PARTITION BY store_number * 1000 + product_code;
```

If many queries specify both `store_number` and `product_code`, this might be a useful partitioning expression. One downside is that it fails if the number of products grows to the point that a four-digit number is required, or if the number of stores expands beyond 64.

Note that the Optimizer assumes 65,535 partitions even though some might be empty. The table is not any larger because of the empty partitions, though the Optimizer default assumption that there are 65,535 partitions based on the specification might sometimes mislead it into making suboptimal plan choices.

Some disadvantages of this form are that the partitioning cannot be altered unless the table is empty and row partition elimination for queries is usually limited to constant or USING value equality conditions on both of the partitioning columns.

An alternative is to use multilevel partitioning, as demonstrated by the following CREATE TABLE request:

```
CREATE TABLE ...
PRIMARY INDEX (store_number, product_code, sales_date)
PARTITION BY (RANGE_N(store_number BETWEEN 0
                        AND      64
                        EACH      1),
              RANGE_N(product_code BETWEEN 1
                        AND      999
                        EACH      1));
```

Basing the Partitioning Expression on a CASE_N Function

You can use the `CASE_N` function to concisely define a partitioning expression for which each partition contains data based on an associated condition. When you specify `CASE_N` for single-level partitioning expression, two partition numbers, NO CASE and UNKNOWN, are automatically reserved for specific uses.

The `CASE_N` function is patterned after the SQL CASE expression (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145). The function returns an INTEGER value numbered from 1, indicating which `CASE_N` condition first evaluated to TRUE for the particular value. The returned value can map directly to a partition number or be further modified to calculate the partition number.

Assume a table has a `total_revenue` column, defined as DECIMAL. The table can be partitioned on that column, so that low revenue products are separated from high revenue products. The partitioning expression can be written as follows:

```
CREATE TABLE ...
PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY CASE_N (total_revenue < 10000,
                    total_revenue < 100000,
```

```
total_revenue < 1000000,
NO CASE, UNKNOWN);
```

This request defines 5 partitions, conceptually numbered from 1 to 5 in the order they are specified in the partitioning expression.

This partition number ...	Represents ...
1	products with total_revenue less than 10,000.
2	products with total_revenue of at least 10,000, but less than 100,000.
3	products with total_revenue of at least 100,000 but less than 1,000,000.
4 (NO CASE)	any value that does not evaluate to TRUE or UNKNOWN for any previous CASE_N condition, which in this case is total_revenue equal to or greater than 1,000,000.
5 (UNKNOWN)	values for which it is not possible to determine the truth value of a previous CASE_N expression. For this partitioning condition, a row with a null for total_revenue is assigned to the UNKNOWN partition, because by definition, it is not possible to evaluate whether a null is less than 10,000.

The rows in the NO CASE and UNKNOWN partitions are valid, and the system accesses those partitions to process queries unless the query conditions exclude them. By defining NO CASE and UNKNOWN partitions, you can ensure that any possible value maps to a partition. In the absence of those partitions, some values result in errors and so are not be inserted into the table. In practice, it is probably better to have more than three revenue ranges unless the queries against this table rarely specify narrower revenue ranges.

This example demonstrates that CASE_N can be used to define complicated partitioning expressions tailored to a specific table and specific query workloads:

```
CREATE TABLE
...
PRIMARY INDEX (col1, col2)
PARTITION BY CASE_N (col3 < 6,
                    col3 >= 8
                    AND col3 < 10
                    AND col4 <> 12,
                    col5 <> 10
                    OR col3 = 20,
                    NO CASE OR UNKNOWN);
```

Without knowing the meaning and data demographics of the columns, there is no way of knowing whether this partitioning expression is useful.

Note that the NO CASE and UNKNOWN partitions are combined into a single partition in this example.

Unlike the case for the previous partitioning expression examples, the Optimizer knows how many partitions are defined when CASE_N is used as the partitioning expression and does not have to assume a default number of 65,535 partitions.

There are several disadvantages to this form:

- The partitioning of the table cannot be altered unless it is empty.
- Row partition elimination for queries is often limited to constant or USING value equality conditions on the partitioning columns.
- The Optimizer might not eliminate some row partitions that it possibly could if the partitioning were better conceived.
- As the number of conditions increases, evaluating a CASE_N function can be costly in terms of CPU cycles, and a CASE_N might also contribute to causing the table header to exceed its maximum size limit. Therefore, you may need to limit the number of conditions you define in the CASE_N function to a relatively small number.

Considerations for Basing a Partitioning Expression on a CASE_N Function

Building a partitioning expression on CASE_N is a reasonable thing to do only if the following items are all true:

- The partitioning expression defines a mapping between conditions and INTEGER numbers.
- There are limited number of conditions (too many conditions can lead to excessive CPU usage or the table header exceeding its maximum size limit).
- Your query workloads against the table use equality conditions on the partitioning columns to specify a single partition.
- You have no need to alter the partitioning of the table.
- You get the plans and data maintenance you need.

Basing a Partitioning Expression on a RANGE_N Function

The RANGE_N function is provided to simplify the specification of common partitioning expressions where each partition contains a range of numeric data, and is especially useful when the column contains a date or timestamp. RANGE_N returns an INTEGER determined by the first range that includes the column value, numbered from 1, that can be mapped directly to a partition number or can be further modified to calculate the partition number. RANGE_N is commonly used to define partitioning expressions. When you use a RANGE_N function to define a single-level partitioning expression, two partition numbers, NO RANGE and UNKNOWN, are reserved for specific uses.

Assume a table with 7 years of order data, ranging from 2001 through 2007. The next partitioning expression creates 84 partitions, one for each month of the period covered by the data:

```
CREATE TABLE
...
PRIMARY INDEX (order_number)
PARTITION BY RANGE_N (order_date BETWEEN DATE '2001-01-01'
```

```

AND      DATE '2007-12-31'
EACH INTERVAL '1' MONTH);

```

Each partition contains roughly the same number of rows, assuming that order volume has stayed roughly constant across the seven year interval. Neither a NO RANGE nor an UNKNOWN partition is defined because this partitioning expression definition is for data that only has dates within the ranges specified in the partitioning expression.

It is frequently desirable to have each partition contain roughly the same number of rows, but it is not required. The next example puts the older orders into partitions with coarser granularity, and the newer orders into partitions with finer granularity:

```

CREATE TABLE
...
PRIMARY INDEX (order_number)
PARTITION BY RANGE_N (order_date BETWEEN DATE '1994-01-01'
                        AND      DATE '1997-12-31'
                        EACH INTERVAL '2' YEAR,
                        DATE '1998-01-01'
                        AND      DATE '2000-12-31'
                        EACH INTERVAL '1' YEAR,
                        DATE '2001-01-01'
                        AND      DATE '2003-12-31'
                        EACH INTERVAL '6' MONTH,
                        DATE '2004-01-01'
                        AND      DATE '2007-12-31'
                        EACH INTERVAL '1' MONTH,
                        NO RANGE, UNKNOWN);

```

In this example, the more recent data is partitioned more finely than the older data. This can be a good strategy if you know that the older data is rarely accessed except as part of a full-table scan, because it reduces some potential disadvantages by defining a smaller number of partitions. However, maintaining this structure over extended epochs of time is not as simple as maintaining a structure in which each interval covers the same time duration. In this example, the years 2004 through 2007 are partitioned by month. As time passes, and 2004 data becomes older and less frequently referenced, it shall become necessary to repartition the table if the pattern of defining longer time intervals for older data is to be maintained.

It is both easy and fast to add and drop partitions from the ends of a partitioning definition, but repartitioning intervals in the middle partitions requires much more work, and it is usually faster to reload the data. In addition, when a range partition is dropped, any rows in that partition are moved from the dropped range partition to the NO RANGE partition or to an added range partition. Also, when a range is added, rows might need to be moved from the NO RANGE partition to the new range partition.

Some expansion room is allowed for future dates by specifying the final partition as extending to the end of 2007. It is easy to add and drop ranges to the end of a partitioning expression using the ALTER TABLE

statement. While you could have specified an ending date far in the future, such as 2099-12-31, it is generally not desirable to define hundreds of partitions that shall not be used for decades.

The example shows intervals of years and months. It is also possible to partition by day (EACH INTERVAL '1' DAY) or by week (EACH INTERVAL '7' DAY). A seven-day interval can start on any day of the week, so if you want to start the weekly intervals on Sunday, for example, the beginning date should be chosen so that it falls on a Sunday. Also, the last range might be less than specified by the EACH clause. For example, suppose that with seven-day intervals the first date falls on Sunday, but the last date also falls on Sunday, in which case the last range spans only one day.

The RANGE_N NO RANGE clause is comparable to the CASE_N NO CASE clause, and the UNKNOWN clause has the same meaning for both functions.

As with the CASE_N partitioning expression, the Optimizer knows how many partitions are defined when RANGE_N is used as the partitioning expression.

Considerations for Basing the Partitioning Expression on a RANGE_N Function

Using the RANGE_N function to build a partitioning expression offers the following advantages:

- Defining an efficient mapping or ranges between integer (BYTEINT, SMALLINT, INTEGER, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP type and INTEGER numbers.
- Provides more opportunities than other expressions to optimize queries.
- The Optimizer knows the number of defined partitions when you specify RANGE_N to define the partitioning expression.

For partitioning expressions other than RANGE_N or CASE_N, the Optimizer generally assumes a total of 65,535 partitions when statistics have not been collected, which could easily be far more than the number of populated partitions. However, collecting statistics on PARTITION can provide information about which partitions are empty.

- Faster partitioning changes than any other expression using the following ALTER TABLE options.
 - ADD RANGE
 - DROP RANGE

You can optimize the effects of using RANGE_N in your partitioning expression by observing the following guidelines:

- Reference only a single integer (BYTEINT, SMALLINT, INTEGER, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP column, not expressions.

For example, specifying a simple expression such as $x/10$ in place of a column name in the RANGE_N specification, even if the expression references only a single column, can hinder, or even prevent, row partition elimination.

- For equal-sized ranges, always specify an EACH clause.

Note the following collateral facts about equal- and unequal-sized partitions:

- The performance of unequal-sized partitions varies depending on which partitions are accessed.
- Unequal size ranges can prevent fast partitioning changes from being made using the ALTER TABLE statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).
- Using the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN specifications for a range can negatively affect later ALTER TABLE partitioning strategies.

Do not use these clauses unless you have specific reasons for doing so (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for details).

Reasons not to use the NO RANGE, NO RANGE OR UNKNOWN, and UNKNOWN clauses include the following.

- The maintenance and use of partitioned tables is simpler if you do not use these options, and avoiding their use also prevents bad data from being inserted into the table.
- If these partitions are not eliminated by row partition elimination, they can cause negative performance impacts if they contain a large number of rows.
- If a partitioning column is NOT NULL such that test values can never be null, do not specify UNKNOWN or NO RANGE OR UNKNOWN.
- If the specified ranges cover all possible values, do not specify NO RANGE or NO RANGE OR UNKNOWN.

Basing the Partitioning Expression on a RANGE_N Character Column

The RANGE_N function provides a simplified method for mapping a character value into one of a list of specified ranges, and then returning the number of that range. A range is defined by its starting and ending boundaries, inclusively.

If you do not specify an ending boundary, the range is defined by its starting boundary, inclusively, up to, but not including, the starting boundary for the next range.

The EACH clause is not supported for character test values, and the system returns an error if you specify an EACH clause in a RANGE_N function with a character test value. Each range specified for expressions character data types maps to exactly an integer range of only one value. Therefore, the number of partitions that can be specified is somewhat limited because of table header limitations and due to the increase in CPU usage to handle a large number of ranges.

You can specify an asterisk for the first starting boundary to indicate the lowest possible value, and you can specify an asterisk for the last ending boundary to indicate the highest possible value.

As with numeric RANGE_N expressions, options are provided to handle cases when the value does not map into one of the specified ranges or evaluates to UNKNOWN because of a null result, making it impossible to determine into which range the value would map.

The following rules apply to the use of the RANGE_N function for character data in addition to the rules that exist for numeric data:

- You can specify only one test value, and it must result in an integer (BYTEINT, INTEGER, SMALLINT, BIGINT), character (CHARACTER, GRAPHIC, VARCHAR, VARCHAR(n) CHARACTER SET GRAPHIC), DATE, or TIMESTAMP data type.
- A RANGE_N partitioning expression can specify the UPPERCASE qualifier and the following functions.
 - CHAR2HEXINT
 - INDEX
 - LOWER
 - MINDEX
 - POSITION
 - TRANSLATE
 - TRANSLATE_CHK
 - TRIM
 - UPPER
 - VARCHAR(n) CHARACTER SET GRAPHIC
- The database aborts the request and returns an error if any of the specified ranges are defined with null boundaries, are not increasing, or overlap. Increasing order is determined using the session collation and the case sensitivity specification for the test value at the time the table is created.

Using CASE_N and RANGE_N in SELECT Requests

You can also use the CASE_N and RANGE_N functions in a SELECT request. For example, you might use them to help determine the distribution of rows among partitions for a proposed partitioning expression. For example, before deciding to partition on division_number, you might want to check the resulting distribution with a request like the following.

```
SELECT RANGE_N(division_number BETWEEN 1
                                AND    4
                                EACH   1) AS p,
        COUNT(*) AS c,
FROM sales
GROUP BY p
ORDER BY p;
```

Another use of RANGE_N is to determine the number of ranges defined, for example, as follows.

```
SELECT RANGE_N(DATE '2007-12-31' BETWEEN DATE '2001-01-01'
                                AND    DATE '2007-12-31'
                                EACH INTERVAL '30' DAY);
```

This query returns the value 86 because the last range is less than 30 days.

The following query over the same data returns the value 84 because the number of days per range varies between 28 and 31, depending on the month and year.


```
SELECT RANGE_N(
  DATE '2007-12-31' BETWEEN DATE '2001-01-01'
    AND DATE '2007-12-31'
    EACH INTERVAL '1' MONTH);
```

The final example returns the value 13 because the last range is only one day in length.

```
SELECT RANGE_N(
  DATE '2002-01-01' BETWEEN DATE '2001-01-01'
    AND DATE '2002-01-01'
    EACH INTERVAL '1' MONTH);
```

Workload Characteristics, Queries, and Row Partition Elimination

Row partition elimination is most effective in the following situations. You should always verify (using the EXPLAIN request modifier) that you are getting the desired results for any plans.

- Row partition elimination is most effective with constant conditions on the partitioning columns.
- When a row partitioning expression is written using something other than the RANGE_N function or a single column, row partition elimination is most effective when you specify constant equality conditions.
- Row partition elimination can also be effective with equality conditions on USING variables if the conditions specify a single partition.
- Row partition elimination occurs for CURRENT_DATE and DATE built-in functions for inequality conditions. This does not prevent the request from being cached.
- Row partition elimination might occur for other built-in functions and USING variables in inequality conditions, and if it does, the action prevents the system from caching the request.
- Multiple ORed equality conditions on the same row partitioning column do not invoke partition elimination.

As an alternative, you should try either to use the UNION operator on two SELECT requests or to substitute constants for the USING variables in any inequality conditions.

- Use simple comparison of a partitioning column to a constant, built-in function, or USING variable expression for your query conditions.

For example:

- `d=10d >= 10 AND d <= 12`
- `d BETWEEN 10 AND 12d = 10+1`
- `d IN (20, 22, 24)d = 20 OR d=21`
- `d = :udd`
- `d BETWEEN CURRENT_DATE-7 and CURRENT_DATE-1`
- Avoid specifying query conditions with expressions or functions constructed on the row partitioning column. For example, use the form in Example 2 rather than the form in Example 1, and the form in Example 4 rather than the form in Example 3.

Example 1

The predicate in this query is based on an expression constructed on the row partitioning column x:

```
CREATE TABLE
...
PARTITION BY x;
SELECT ...
WHERE x+1 IN (2,3);
```

Example 2

The predicate in this query is based only on the value of the row partitioning column x:

```
CREATE TABLE
...
PARTITION BY RANGE_N(x BETWEEN 1
                      AND 65533
                      EACH 1);
```

Note:

In this example, it is preferable to specify the exact upper limit of the range of x, if it is less than 65,533, in this CREATE TABLE request rather than 65,533. Use the max value of x instead of 65,553.

```
SELECT ...
WHERE x IN (1,2);
```

Example 3

The predicate in this query is based only on the value of row partitioning column x even though the table is partitioned by the expression x + 1:

```
CREATE TABLE
...
PARTITION BY x+1;
SELECT ...
WHERE x IN (1,2);
```

Example 4

The predicate in this query is based only on the value of row partitioning column x:

```
CREATE TABLE
...
```

```
PARTITION BY RANGE_N(x BETWEEN 0
                      AND 65532
                      EACH 1);
```

Note:

In this example, it is preferable to specify the exact upper limit of the range of x, if it is less than 65,552, in this CREATE TABLE request rather than 65,532. Use the max value of x instead of 65,532

```
SELECT ...
WHERE x IN (1,2);
```

Workload Characteristics and Row Partitioning

Consider using row partitioning when your workloads have any of the following characteristics:

- The number of queries in workloads that access the table has a range constraint, particularly a date constraint on some column of the table.
- Queries have an equality constraint on some column of the table, and that column is either not the only primary index column or it is not a primary index column at all.
- If there is a primary index that is used only, or principally, to achieve an even distribution of rows, but not usually for accessing or joining rows, and access is frequently made on a column that is suitable for partitioning.
- If there is a primary index that is used to achieve an even distribution of rows as well as for accessing or joining rows, and columns suitable for partitioning are included in the primary index definition.
- If there is a primary index that is used to achieve an even distribution of rows as well as for accessing or joining rows, but columns suitable for partitioning are not included in the primary index definition. This might be a good candidate for partitioning, but you need to pay particular attention to weighing the performance tradeoffs that often result in this situation.
- Use the RANGE_N function for a partitioning expression, preferably on a column with a DATE or TIMESTAMP data type, because it generally provides more opportunities for row partition elimination, and the Optimizer knows the exact number of defined row partitions.

For instance, instead of the following row partitioning expression.

```
PARTITION BY column
```

use

```
PARTITION BY RANGE_N(column BETWEEN m
                      AND      n
                      EACH    s)
```

Dates and timestamps are often used in query conditions and therefore makes good candidates for a partitioning expression.

When partitioning on a DATE column, use RANGE_N with a single overall range divided into ranges of equal size as follows.

```
PARTITION BY RANGE_N(date_column BETWEEN DATE '...'
                        AND      DATE '...'
                        EACH INTERVAL 's' t)
```

- Use DATE constants or TIMESTAMP constants such as DATE '2011-08-06' or TIMESTAMP '2011-08-25 10:14:59' to specify the ranges in a partitioning expression. This is not only easier to read, making it clear that the expression is defined on a date, but it also removes the dependence on the FORMAT used in the implicit conversion to a date. You can also specify TIMESTAMP(n) WITH TIME ZONE constants for RANGE_N partitioning.

For example, you can specify a RANGE_N-based partitioning expression like the following.

```
RANGE_N(ts BETWEEN TIMESTAMP '2003-01-01 00:00:00+13:00'
        AND      TIMESTAMP '2009-12-31 23:59:59-12:59'
        EACH INTERVAL '1' MONTH)
```

Use an INTERVAL constant in the EACH clause where the variable *t* is DAY, MONTH, YEAR, or YEAR TO MONTH.

Do not use INTEGER values or CHARACTER constants for dates in your partitioning expressions, since they can easily be incorrectly specified such that they do convert to the date you expected.

For example, it might seem intuitive to simply partition by the name of a DATE column as follows:

```
PARTITION BY sales_date
```

This form does not produce a syntax error. In fact, it works correctly only for dates in the early 1900s and follows the rule of implicit conversion to get an INTEGER partition number, but such a table is not generally useful.

For this case, you should instead use RANGE_N with a granularity of EACH INTERVAL '1' DAY.

It might also seem intuitive to specify something like the following to indicate that a date column in the primary index is to be partitioned by week:

```
PARTITION BY 7
```

However, this form does produce a syntax error.

For this case, use RANGE_N with a granularity of EACH INTERVAL '7' DAY.

- Consider specifying only as many date ranges as are currently needed plus a few additional ranges for the future.

By limiting ranges to those that are currently needed, you help the Optimizer to better cost plans and also allow for more efficient primary index access, joins, and aggregations when the partitioning column is not included in the primary index.

This is not as important if you collect current PARTITION statistics.

A good guideline is to define 10% or fewer of the partitions to be empty to be able to handle future dates. You should also define enough future ranges to minimize the frequency of ALTER TABLE statements needed to drop and add ranges. However, if you make changes too infrequently, you might forget to alter the table entirely.

Altering the table only once a year is probably not a good idea for the following reasons:

- Because you must create too many empty partitions.
- Because it is too easy to forget to alter the partitioning ranges if you do not do it fairly regularly.
- Because you fail to follow the procedure often enough to prevent problems from occurring when you finally get around to following it.

You must balance these concerns of having enough future partitions, but not too many.

- RANGE_N permits a faster partitioning change using ALTER TABLE ... DROP RANGES or ALTER TABLE ... ADD RANGES.
- Reference a single INTEGER or DATE column in the RANGE_N function.

Do not use an expression such as x/10 in place of a simple column reference in a partitioning expression constructed from a RANGE_N function even if the expression only references a single column.

- Define ranges of equal size using EACH to specify the granularity of the partition.

Multiple ranges, with or without specifying an EACH granularity, require more CPU time to execute and different sized ranges can prevent fast partitioning changes.

- Consider not specifying the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions in the RANGE_N function.

Using these partitions can degrade query performance because queries can be forced to scan data in these partitions unnecessarily. If you specify the NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN partitions, you can also affect the performance of ALTER TABLE partitioning change negatively because data might need to be moved among the partitions. An UNKNOWN partition is not needed if the partitioning cannot produce rows with unknown partition values (this is often the case when a partitioning column is specified to be NOT NULL).

- Deciding not to use a RANGE_N function for a partitioning expression can be a good choice in the following instances:
 - You do not partition the table or join index on a DATE column.
 - You use constant or USING variable equality conditions on the partitioning columns in a majority of the queries in your workloads to specify a single partition.

If the assumption made by the Optimizer that the table (not based on a CASE_N partitioning expression) has 65,535 partitions provides good plans in these cases, you do not need to alter partitioning, and the system produces the plans and data maintenance performance your site requires.

Workload Characteristics and Partitioning

Consider the following points when you define the partitioning for a table or join index:

- You must choose a primary index, primary AMP index, or no primary index to achieve an even distribution of rows to the AMPs.
- When appropriate, the primary index or primary AMP index column set ought to be constructed from columns that are often constrained by equality conditions in queries in order to provide fast access, joins, and aggregations on those columns.

If the partitioning column set is included in the primary index column set, some concerns are not an issue. However, you would not add the partitioning columns to the primary index to avoid these concerns anyway, because there is usually little, if any, benefit in doing so.

- If the partitioning columns are included in the set of primary index columns, then you can specify the primary index to be UNIQUE.

Adding a partitioning column as a primary index column just to make the primary index unique is not effective, however, because it is the original set of primary index columns that needs to be unique and possibly used for access, joins, and aggregations.

- If the partitioning columns are also included in the set of primary index columns, it might be a good choice to define many combined partitions for primary index access and joins. Be aware that plan costing can be affected if there are too many empty combined partitions if PARTITION statistics are not collected. Other factors can also lead to having fewer combined partitions.
- If all of the partitioning columns are not included in the primary index column set, the primary index cannot be defined as a UPI.

If the primary index or primary AMP index columns must be unique, define a USI on them.

Because MultiLoad and FastLoad do not support USIs, you must use another load strategy such as any of the following:

- Loading the rows using Teradata Parallel Data Pump (see *Teradata® Parallel Data Pump Reference*, B035-3021).
- Loading the rows into a staging table followed by an INSERT ... SELECT or MERGE into the target table using error logging (see the information about CREATE ERROR TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and the information about INSERT ... SELECT and MERGE in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).
- Dropping the USI, loading the rows using MultiLoad or FastLoad, and then recreating the USI (see *Teradata® FastLoad Reference*, B035-2411 and *Teradata® MultiLoad Reference*, B035-2409).
- Including the partitioning columns in the PI, noting the previously documented problems with this approach.

You must evaluate the tradeoffs among these choices carefully.

- If the primary index or primary AMP index columns need to be an efficient access path and there are many combined partitions, consider one of these options:

- Defining a USI on the primary index columns to improve access time.
- Defining a NUSI on the primary index columns to improve access time.
- Creating a join index to cover queries made against the table.
- Creating a hash index to cover queries made against the table.
- Consider defining fewer combined partitions when a table is also accessed or joined on the primary index.

Workload Characteristics and Joins

Consider the following points when you write join queries against a partitioned table:

- Specify equijoins on the primary index and partitioning column sets, if possible, in order to prejudice the Optimizer to use efficient RowKey-based joins (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142).

Consider including the partitioning column in the nonpartitioned primary index table so you can join on the partition column. This means that, depending on the situation, you might want to consider denormalizing the physical schema to enhance the performance of partitioned table-to-nonpartitioned table joins.

- If you specify an equijoin on the primary index column set, but not on the partitioning column set, the fewer combined partitions that exist after any row partition elimination, the better.

Otherwise, the table might need to be spooled and sorted.

The Optimizer can specify sliding-window joins when there are a small number of participating combined row partitions (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142).

- Use RANGE_N to define fewer partitions and specify conditions on the row partitioning columns to reduce the number of combined row partitions involved in the join by evoking partition elimination.

To ensure that the Optimizer creates good query plans for your partitioned tables, you should always collect PARTITION statistics and keep them current.

If you have not collected PARTITION statistics, the Optimizer does not know whether a combined row partition is empty or not, so it has to assume all defined combined row partitions might have rows with respect to the plan it generates; however, it might choose among several such plans based on the estimated number of populated combined row partitions.

- Dynamic row partition elimination for a product join improves performance when a partitioned table and another table are equijoined on the row partitioning column of the partitioned table (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142).

Remember to collect statistics (see the information about COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144) on all of the following.

- The primary indexes of both tables.
- The partitioning columns of the partitioned table.

- The column in the nonpartitioned table that is equated to the partitioning column of the partitioned table.
- The system-derived PARTITION column of all partitioned tables.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

General Recommendations for Using Row-Partitioned Tables and Join Indexes

To take optimal advantage of row-partitioning as it is used in Vantage for row-partitioned tables and join indexes, you must have a thorough understanding of partitioning expressions, the general notion of partitioning, and the specific attributes that partitioning brings to a table. The placement of data on the AMPs and the use of row partition elimination by the system can significantly improve the performance of some queries, while at the same time degrading the performance of other queries. You must consider the impact of partitioning on data maintenance. You must be aware that partitioning increases the size of each row header in a partitioned table or index by either 2 or 8 bytes (partitioned table rows are 4 bytes wider if multivalued compression is specified for the table) and that partitioning also increases the size of each secondary index row by 2 or 8 bytes for each referencing ROWID in the index.

Partitioning is a physical database design consideration, and like any other physical database design issue, it is more likely to work well if you have done a good logical database design first.

You should not focus on any one aspect of partitioning while undertaking the process of creating the physical design for your databases. Each and every one of the following attributes must work well together to ensure the success of your partitioned tables and join indexes.

- The partitioning expression
- Queries

This includes both those queries designed specifically to access the partitioned table and the general class of all queries that are likely to access it.

- Performance, access methods, join strategies, row partition elimination
- Ease of altering the partitioning expression
- Effects of the row partitioning on data maintenance for the table
- Backup and restore operations on the table

A successful partitioning expression is one that takes advantage of row partition elimination, supports ease of partition altering with ALTER TABLE, and has no significant negative impact on data maintenance.

You should always experiment with your intended uses of partitioned tables, considering and analyzing performance tradeoffs between using partitioning or not, various partitioning strategies, and using partitioning along with, or instead of, other indexing such as secondary, hash, and join indexes.

Analyze the maintenance process choices and their performance. Note that additional maintenance is required if you define a USI to enforce uniqueness on a column set of a partitioned table.

Finally, you must always ensure that you are getting the results that you expect. Be sure to review EXPLAIN reports, looking for row partition elimination and rowkey-based joins. Defining a sophisticated partitioning

expression is helpful only if the queries in your workloads are able to invoke row partition elimination. Be sure to measure performance for the query workload and for critical queries both before and after creating the partitioned table or join index. Never assume that partitioning will improve the performance of your maintenance workloads, verify it. And always weigh the costs against the benefits.

Single-Level Partitioning

Partitioning CHECK Constraints for Single-Level Partitioning

Vantage derives a table-level partitioning CHECK constraint from the partitioning expression. The text for this derived partitioning constraint cannot exceed 16,000 characters; otherwise, the system aborts the request and returns an error to the requestor.

The following diagrams provides the two forms of this partitioning CHECK constraint derived for single-level partitioning. The forms differ depending on whether the partitioning expression has an INTEGER data type or not.

The first form applies to partitioning expressions that do not have an INTEGER type. Call this partitioning constraint form 1.

```
CHECK ((CAST((partitioning_expression) AS INTEGER)) BETWEEN 1 AND max)
```

The second form applies to partitioning expressions that have an INTEGER type. Call this partitioning constraint form 2.

```
CHECK ((partitioning_expression) BETWEEN 1 AND max)
```

partitioning_expression

Partition number returned by the single-level partitioning expression.

max

Maximum number of partitions defined by *partitioning_expression*. If *partitioning_expression* is defined by the RANGE_N or CASE_N function, *max* is the number of partitions defined by the RANGE_N or CASE_N function. Otherwise, *max* is 65535.

Multilevel-partitioned tables have a different table-level partitioning CHECK constraint (see [Multilevel Partitioning](#)).

If any one of the following items is true, Vantage uses a different form of partitioning constraint:

- A partitioning expression for one or more levels consists solely of a RANGE_N function with a BIGINT data type.
- An ADD clause is specified for one or more partitioning levels.
- The table has 8-byte partitioning.
- There is a column-partitioning level.

- For other than level 1 of a populated table, the number of partitions for at least one level changes when the table is altered.
- For other than level 1 of an empty table, the number of partitions for at least one level decreases when the table is altered.

The format for this partitioning constraint text is as follows. Call this partitioning constraint form 4.

```
CHECK (/*nn bb cc*/ partitioning_constraint_1 ...
      [ AND partitioning_constraint_n ])
```

nn

The number of partitioning levels:

- For 2-byte partitioning, *nn* ranges between 01 and 15, inclusive.
- For 8-byte partitioning, *nn* ranges between 01 and 62, inclusive.

bb

The number of bytes used to store the internal partition number in the row header:

- For 2-byte partitioning, *bb* = 2.
- For 8-byte partitioning, *bb* = 8.

cc

The column partitioning level:

- If there is no column partitioning, *cc* = 00.
- If there is column partitioning, *cc* ranges between 01 and *nn*, inclusive.

partitioning_expression_i

The partitioning expression at partitioning level *i*, where *i* is in [1, *nn*]. Leading zeros are not used for the value of *i*.

partitioning_constraint_i

- *partitioning_constraint_i* /*i d+a*/ IS NOT NULL
if there is row partitioning at partitioning level *i*.
- PARTITION#Li /*i d+a*/ = 1
if there is column partitioning at partitioning level *i*.

The partitioning constraint at partitioning level *i*, where *i* is in [1, *nn*]. Leading zeros are not used for the value of *i*.

d is the number of currently defined partitions for the level. For a column-partitioned level, this includes the 2 internal column partitions. Leading zeros are not used for the value of *d*.

a is the number of additional partitions that could be added. Leading zeros are not used for the value of *a*.

Each of the partitioning constraints corresponds to a level of partitioning in the order defined for the table.

The TableCheck column of DBC.TableConstraints contains the unresolved condition text for a table-level CHECK constraint check or implicit table-level constraint such as a partitioning constraint. The ConstraintType code for such a partitioning constraint is Q for a partitioned object. For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

Rows that violate this partitioning CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

Assume you have created the following table.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey  BETWEEN 0
                           AND 49999
                           EACH 100))
UNIQUE INDEX (o_orderkey);
```

The partitioning CHECK constraint SQL text that is stored in *DBC.TableConstraints* for this multilevel partitioned primary index is as follows.

```
CHECK(RANGE_N(o_custkey BETWEEN 0
                           AND 49999
                           EACH 100)
BETWEEN 1 AND 500)
```

Now suppose that you create a column-partitioned version of the *orders* table, *orders_cp*, with the following definition.

```
CREATE TABLE orders_cp (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
```

```

o_orderstatus CHARACTER(1) CASESPECIFIC,
o_totalprice  DECIMAL(13,2) NOT NULL,
o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL,
o_comment     VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (RANGE_N o_custkey BETWEEN 0
              AND 100000
              EACH 1), COLUMN)
UNIQUE INDEX (o_orderkey);

```

The partitioning CHECK constraint for this table with 8-byte partitioning is as follows.

```

CHECK (/*02 08 02*/ RANGE_N(o_custkey BETWEEN 0 AND 100000 EACH
1) /*1 100001+485440633518572409*/ IS NOT NULL AND PARTITION#L2 /
*2 9+10*/ =1)

```

You could use the following SELECT request to retrieve the level for the column partitioning for each of the objects that have column partitioning in the system.

```

SELECT DBaseId, TVMId, ColumnPartitioningLevel (TITLE
'Column-Partitioning Level')
FROM DBC.TableConstraints
WHERE ConstraintType = 'Q'
AND ColumnPartitioningLevel >= 1
ORDER BY 1,2;

```

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details about *DBC.TableConstraints* and its role in recording partitioning metadata.

The maximum size of all partitioning CHECK constraints is 16,000 characters.

Single-Level Partitioning Example

The following multipart example demonstrates the various properties of different single-level partitioning of the same data.

Stage 1: First single-level partitioning of the *orders* table.

```

CREATE TABLE orders (
o_orderkey INTEGER NOT NULL,
o_custkey  INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 0
                    AND 100
                    EACH 10); /* p1 */

```

This definition implies the following information about the partitioning of orders:

- Number of partitions in the first, and only, level = $d_1 = 11$
- Total number of combined partitions = $d_1 = 11$
- Combined partitioning expression = p_1

If the value of `o_custkey` is 15, then the following additional information is implied:

- Partition number for level 1 = $\text{PARTITION\#L1} = p_1(15) = 2$.
- PARTITION\#L2 through PARTITION\#L15 are all 0.
- Combined partition number = $\text{PARTITION} = p_1(15) = 2$.

Value of <code>o_custkey</code>	Result of the <code>RANGE_N</code> function Value of <code>PARTITION</code> Value of <code>PARTITION\#L1</code>
0 - 9	1
10 - 19	2
20 - 29	3
30 - 39	4
40 - 49	5
50 - 59	6
60 - 69	7
70 - 79	8
80 - 89	9
90 - 99	10
100	11

Stage 2: Second single-level partitioning of the orders table.

Suppose you then submit the following `ALTER TABLE` request on orders:

```
ALTER TABLE orders
  MODIFY PRIMARY INDEX
    DROP RANGE BETWEEN 0
      AND      9
    EACH      10;
```

This alters the partitioning expression to:

```
RANGE_N(o_custkey BETWEEN 10
        AND      100
        EACH     10);
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 10
                     AND      100
                     EACH     10); /* p1 */
```

This changes the information implied by the initial table definition about the partitioning of orders as follows:

- Number of partitions in the first, and only, level = $d_1 = 10$
- Total number of combined partitions = $d_1 = 10$
- Combined partitioning expression = p_1

Now if *o_custkey* is 15, the following additional information is implied:

- Partition number for level 1 = PARTITION#L1 = $p_1(15) = 1$.
- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION = $p_1(15) = 1$.

The following table indicates the new partition numbers for the various defined ranges for *o_custkey*:

Value of o_custkey	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
10 - 19	1
20 - 29	2
30 - 39	3
40 - 49	4
50 - 59	5
60 - 69	6
70 - 79	7
80 - 89	8
90 - 99	9

Value of o_custkey	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
100	10

Stage 3: Third single-level partitioning of the orders table.

Suppose you submit the following ALTER TABLE request on orders:

```
ALTER TABLE orders
  MODIFY PRIMARY INDEX
  ADD RANGE BETWEEN 5
        AND      9
        EACH     1;
```

This alters the partitioning expression to:

```
RANGE_N(o_custkey BETWEEN 5
        AND      9
        EACH     1, 10 AND 100
        EACH     10);
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY
RANGE_N(o_custkey BETWEEN 5
        AND      9
        EACH     1, 10 AND 100
        EACH     10); /* p1 */
```

This changes the information implied by the second table definition about the partitioning of orders as follows:

- Number of partitions in the first, and only, level = $d_1 = 15$
- Total number of combined partitions = $d_1 = 15$
- Combined partitioning expression = p_1

Now if *o_custkey* is 15, the following additional information is implied:

- Partition number for level 1 = PARTITION#L1 = $p_1(15) = 6$.

- PARTITION#L2 through PARTITION#L15 are all 0.
- Combined partition number = PARTITION = $p_1(15) = 6$.

The following table indicates the new partition numbers for the various defined ranges for *o_custkey*.

Value of <i>o_custkey</i>	Result of the new RANGE_N function	
	Value of PARTITION	Value of PARTITION#L1
5		1
6		2
7		3
8		4
9		5
10 - 19		6
20 - 29		7
30 - 39		8
40 - 49		9
50 - 59		10
60 - 69		11
70 - 79		12
80 - 89		13
90 - 99		14
100		15

Note that this table describes the PARTITION#L *n* values for a table having a 2-byte ROWID. If the table had an 8-byte ROWID, there would be as many as 62 partitions.

Stage 4: Fourth single-level partitioning of the orders table.

Suppose you submit the following ALTER TABLE request on orders:

```
ALTER TABLE orders
  MODIFY PRIMARY INDEX
  ADD RANGE BETWEEN 0
            AND      4
            EACH     2;
```

This alters the partitioning expression to be:


```

RANGE_N(o_custkey BETWEEN 0
          AND      4
          EACH    2, 5 AND      9
          EACH    1, 10 AND    100
          EACH    10);

```

In other words, the table definition after you have performed the ALTER TABLE request is as follows:

```

CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_custkey BETWEEN 0
                      AND      4
                      EACH    2, 5 AND 9
                      EACH    1, 10 AND 100
                      EACH    10);

```

This changes the information implied by the third table definition about the partitioning of orders as follows:

- Number of partitions in the first, and only, level = $d_1 = 18$
- Total number of combined partitions = $d_1 = 18$
- Combined partitioning expression = p_1

Now if $o_custkey$ is 15, the following additional information is implied.

- Partition number for level 1 = $PARTITION\#L1 = p_1(15) = 9$.
- $PARTITION\#L2$ through $PARTITION\#L15$ are all 0.
- Combined partition number = $PARTITION = p_1(15) = 9$.

The following table indicates the new partition numbers for the various defined ranges for $o_custkey$.

Value of $o_custkey$	Result of the new RANGE_N function Value of PARTITION Value of PARTITION#L1
0 - 1	1
2 - 3	2
4	3
5	4
6	5
7	6

Value of o_custkey	Result of the new RANGE_N function	
	Value of PARTITION	Value of PARTITION#L1
8		7
9		8
10 - 19		9
20 - 29		10
30 - 39		11
40 - 49		12
50 - 59		13
60 - 69		14
70 - 79		15
80 - 89		16
90 - 99		17
100		18

Single-Level Partitioning Case Studies

This topic presents several case study scenarios to evaluate various approaches to row partitioning a primary index on one level in such a way that optimal benefit is realized.

- The first scenario (see [Scenario 1](#)) is set in a retail sales environment. The effects of row partitioning a sales table on months is examined from several different perspectives.
- The second scenario (see [Scenario 2](#)) is a continuation of the first and attempts to boost the benefits of using a partitioned table further by partitioning on days rather than months. The concept is to investigate how the number of row partitions affects query workload performance.
- The third scenario (see [Scenario 3](#)) is set in a telecommunications environment. The effects of partitioning a telephone call tracking table on call date and phone number are compared and contrasted for the same query workloads.
- The final scenario (see [Scenario 4](#)) illustrates how the decision whether to use row partitioning is often multidetermined, requiring a solution based on numerous, carefully weighted factors.

Determining an Optimal Partitioning Scheme

An optimal partitioning scheme for any table depends on the anticipated query mix. Use your extended logical data model as the starting point for making the decision, but you should always test a few different scenarios to ensure the best partitioning scheme for your particular application and system configuration.

Note that these scenarios are intended to make specific points about partitioning and are not meant to be taken as industry-specific partitioning recommendations.

The following tables list the results of a test prototype performed to examine the potential improvements of using partitioning instead of not using partitioning for the same table.

Test Description	Reduction in Elapsed Time (percent)
Select all rows with a particular value of the partitioning column (200 partitions with roughly the same number of rows each).	98
Select a month of activity from one partition containing six months of data (11 years of data contained in 40 partitions of unequal size).	97
Delete all rows with a particular value of the partitioning column (200 partitions of equal size).	99
Update one column in each row that has a particular value for the partitioning column (200 partitions of equal size).	98

Test Description	Improvement
Teradata Parallel Transporter update operation to insert a number of rows equal to 1% of the table cardinality into one partition out of 200 total partitions.	More than 10 times faster.
Teradata Parallel Transporter update operation to insert a number of rows equal to 1% of the table cardinality into one partition out of 200 total partitions with one NUSI defined on the table.	More than 6 times faster.

Multilevel Partitioning

Multilevel partitioning allows each partition at a given level to be further partitioned into subpartitions. Each partition for a level is subpartitioned the same per a partitioning expression or by column partitioning defined for the next lower level. Rows are grouped by the combined partition number, and within a group are ordered first by hash value and then, if for a partitioned primary index, by uniqueness value. The combined partition numbers are mapped 1-to-1 with internal partition numbers on which the rows on an AMP are ordered. The file system orders rows by the internal partition number, rowhash value, and then uniqueness value.

For multilevel partitioning with a primary index, Vantage hash orders the rows within the lowest partition level. Multilevel partitioning undertakes efficient searches by using row partition elimination at the various levels or combinations of levels. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for a description of row partition elimination and its various forms.

The following list describes the various access methods that are available when multilevel partitioning is defined for a table.

- If there is an equality constraint on the primary index and there are constraints on the partitioning columns such that access is limited to a single partition at each level, access is as efficient as with a nonpartitioned table.

This is a single-AMP, single-hash access in a single subpartition at the lowest level of the partition hierarchy.

- With constraints defined on the partitioning columns, performance of a primary index access can approach the performance of a nonpartitioned primary index depending on the extent of row partition elimination that can be achieved.

This is a single-AMP, single-hash access in multiple (but not all) subpartitions at the lowest level of the partition hierarchy.

- Access by means of equality constraints on the primary index columns that does not also include all the partitioning columns, and without constraints defined on the partitioning columns, might not be as efficient as access with a nonpartitioned primary index. The efficiency of the access depends on the number of populated subpartitions at the lowest level of the row partition hierarchy.

This is a single-AMP, single-hash access in all subpartitions at the lowest level of the partition hierarchy.

- With constraints on the partitioning columns of a partitioning expression such that access is limited to a subset of, say n percent, of the partitions for that level, the scan of the data is reduced to about n percent of the time required by a full-table scan.

This is an all-AMP scan of only the non-eliminated row partitions for that level. This allows multiple access paths to a subset of the data: one for each partitioning expression.

If constraints are defined on partitioning columns for more than one of the partitioning expressions in a multilevel partitioning definition, row partition elimination can lead to even less of the data needing to be scanned.

Partitioning CHECK Constraint for Multilevel Partitioning

A multilevel partitioned table has the following partitioning CHECK constraint, which the system stores in DBC.TableConstraints. Call this partitioning constraint form 3.

```
CHECK (/* nn */ constraint [...])
```

constraint

```
partitioning_expression IS NOT NULL
```

nn

Number of levels, or number of partitioning expressions, in the multilevel partitioning. *nn* can range between 02 and 62, inclusive.

partitioning_expression

Multilevel partitioning level.

Single-level partitioned tables have a different implied table-level partitioning CHECK constraint (see [Single-Level Partitioning](#)).

You can use the following query to retrieve a list of tables and join indexes that have PPIs and their partitioning constraint text.

```
SELECT DatabaseName, TableName (TITLE 'Table/Join Index
Name'),          ConstraintText
FROM DBC.IndexConstraintsV
WHERE ConstraintType = 'Q'
ORDER BY DatabaseName, TableName;
```

You can use a query like the following to retrieve partitioning constraint information for each of the multilevel partitioned objects.

```
SELECT *
FROM DBC.TableConstraints
WHERE ConstraintType = 'Q'
AND SUBSTRING(TableCheck FROM 1 FOR 13) >= 'CHECK (/ *02* /'
AND SUBSTRING(TableCheck FROM 1 FOR 13) <= 'CHECK (/ *15* /';
```

The TableCheck column of DBC.TableConstraints contains the unresolved condition text for a table-level constraint check or implicit table-level constraint such as a partitioning constraint. The ConstraintType code for such an implicit table-level constraint is Q for an object with a partitioned primary index, where in the case of a multilevel partitioning, each of the partitioning levels for the index appears once in the order defined for the table in the text contained in TableCheck. For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

Rows that violate this implied index CHECK constraint, including those whose partitioning expression evaluates to null, are not allowed in the table.

Assume you create the following table:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
```

```

o_comment      VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey  BETWEEN 0
                        AND 49999
                        EACH 100),
              RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                        AND      DATE '2006-12-31'
                        EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);

```

The partitioning CHECK constraint SQL text that would be stored in DBC.TableConstraints for this multilevel partitioned primary index is as follows.

```

CHECK (/*02*/ RANGE_N(o_custkey  BETWEEN 0
                        AND 49999
                        EACH 100)
       IS NOT NULL
AND RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                        AND      DATE '2006-12-31'
                        EACH INTERVAL '1' MONTH)
       IS NOT NULL )

```

The maximum size of this partitioning CHECK constraint is 16,000 characters.

Partitioning CHECK Constraints

A partitioned table or join index has the following partitioning CHECK constraint.

```

CHECK (/*nn bb cc*/ partitioning_constraint_1
       [ AND partitioning_constraint_n ] ... )

```

nn

The number of partitioning levels.

- For 2-byte partitioning, *nn* ranges between 01 and 15, inclusive.
- For 8-byte partitioning, *nn* ranges between 01 and 62, inclusive.

bb

The type of partitioning.

- For 2-byte partitioning, *bb* is 02.
- For 8-byte partitioning, *bb* is 08.

cc

The column partitioning level.

- For no column partitioning, *cc* is 00.
- Otherwise, *cc* ranges between 01 and *nn*, inclusive.

partitioning_constraint_i

Each of the partitioning constraints corresponds to a level of partitioning in the order defined for the table or join index.

The constraint specified by *partitioning_constraint_i* can be the partitioning constraint for any level and can represent either a row partitioning expression or a column partitioning expression.

When *partitioning_expression_i* is the *row partitioning* expression at level *i*, and partitioning constraint is the following.

```
partitioning_expression_i /*i d+a*/ IS NOT NULL
```

When *partitioning_expression_i* is the *column partitioning* expression at level *i* is the following.

```
PARTITION#Li /*i d+a*/ =1
```

a is the number of additional partitions that could be added. Leading zeros are not specified.

Assume you create the following table:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_comment       VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (RANGE_N(o_custkey BETWEEN  0
                        AND 100000
                        EACH      1),
              COLUMN)
UNIQUE INDEX (o_orderkey);
```

The table-level partitioning CHECK constraint SQL text for this 8-byte partitioning is as follows:

```
CHECK (/*02 08 02*/ RANGE_N(o_custkey BETWEEN  0
                        AND 100000
```

```

EACH      1
/*1 100001+485440633518572409*/
IS NOT NULL AND PARTITION#L2 /*2 9+10*/ =1)

```

The maximum size of this table-level CHECK constraint is 16,000 characters.

You can use the following request to retrieve the level for the column partitioning for each of the objects that have column partitioning in the system.

```

SELECT DBaseId, TVMId, ColumnPartitioningLevel
      (TITLE 'Column Partitioning Level')
FROM DBC.TableConstraints
WHERE ConstraintType = 'Q'
AND   ColumnPartitioningLevel >= 1
ORDER BY DBaseId, TVMId;

```

See [Single-Level Partitioning](#) and [Partitioning CHECK Constraint for Multilevel Partitioning](#) for information about the table-level CHECK constraints that Vantage creates for single-level and multilevel partitioned primary index tables and join indexes.

Row Partition Elimination With Multilevel Partitioning

The following examples taken from various industries present examples of how multilevel partitioned primary indexes and row partition elimination can greatly enhance the performance of a query workload.

Example From the Insurance Industry

There are many cases for which row partition elimination using multiple expressions for WHERE clause predicate filtering can enhance query performance (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details about the various forms of row partition elimination). For example, consider an insurance company that frequently performs an analysis for a specific state and within a date range that constitutes a relatively small percentage of the many years of claims history in its data warehouse.

If an analysis is being performed only for claims filed in Connecticut, only for claims filed in all states in June 2005, or only for Connecticut claims filed during the month of June 2005, a partitioning of the data that allows elimination of all but the desired claims should deliver a dramatic performance advantage.

The following example shows how a claims table could be partitioned by a range of claim dates and subpartitioned by a range of state identifiers using multilevel partitioning.

Consider the following multilevel PPI table definition:

```

CREATE TABLE claims (
  claim_id    INTEGER NOT NULL,
  claim_date  DATE NOT NULL,
  state_id    BYTEINT NOT NULL,

```



```

    claim_info VARCHAR(20000) NOT NULL)
PRIMARY INDEX (claim_id)
PARTITION BY (RANGE_N(claim_date BETWEEN DATE '1999-01-01'
                        AND      DATE '2005-12-31'
                        EACH INTERVAL '1' MONTH),
              RANGE_N(state_id BETWEEN 1
                        AND      75
                        EACH 1))
UNIQUE INDEX (claim_id);

```

Eliminating all but one month out of their many years of claims history would facilitate scanning of less than 5% of the claims history (because of business growth, there are many more recent than past claims) for satisfying the following query:

```

SELECT *
FROM claims
WHERE claim_date BETWEEN DATE '2005-06-01' AND DATE '2005-06-30';

```

Similarly, eliminating all but the Connecticut claims from the many states in which this insurance company does business would make it possible to scan less than 5% of the claims history to satisfy the following query:

```

SELECT *
FROM claims, states
WHERE claims.state_id = states.state_id
AND   states.state = 'Connecticut';

```

State selectivity varies by the density of their business book. The company has more business in Connecticut than, for example Oregon and, therefore, a correspondingly larger number of claims for Connecticut. Note that the partitioning in the example specifies up to 75 state_id values to allow for any of the 50 US states plus US territories and protectorates such as Puerto Rico, Guam, American Samoa, the Marshall Islands, Federated States of Micronesia, Northern Mariana Islands, Palau, and the American Virgin Islands in the future.

Combining both of these predicates for row partition elimination makes it possible to scan less than 0.5% of the claims history to satisfy the following query.

```

SELECT *
FROM claims, states
WHERE claims.state_id = states.state_id
AND   states.state = 'Connecticut'
AND   claim_date BETWEEN DATE '2005-06-01' AND DATE '2005-06-30';

```

For evenly distributed data, you would expect scanning of less than 0.25% of the data; however, the data for this company is not evenly distributed among states and dates, leading to a higher percentage of data to scan for the query.

Clearly, combining both predicates for row partition elimination has a significant performance advantage. Row partition elimination by both of these columns, as described, provides higher performance, more space efficiency, and more maintenance efficiency than a composite NUSI or join index for most of the queries run at this insurance company.

In fact, the performance advantage described previously could theoretically be achieved with the current single-level partitioning by using a single partitioning expression that combines both state and monthly date ranges. There are issues with using such a complex, single partitioning expression in this scenario but it is possible to do so.

The bigger problem is that a significant portion of the workload at this insurance company does not specify equality conditions on both partitioning columns. Though theoretically possible, Vantage is currently unable to evaluate more complex conditions with such a partitioning expression. With single-level partitioning, you would have to choose between partitioning by state or partitioning by date ranges.

If you chose partitioning by state, and a user submits a query that specifies a narrow date range without a state filter in the WHERE clause, the system does not perform row partition elimination. At this insurance company, both state-level (and, in some cases, also for a specific date range) analysis and enterprise-level (all states, but for a specific date range) analyses are common.

The users performing state-level analysis do not want to be penalized by having their data combined with all other states, but the users performing enterprise-level analyses also want their queries to be reasonably high-performing, at least by getting date range elimination, and be easy to construct.

Example from the Retail Industry

An example for a large retailer is similar to the insurance example, substituting division for state. In this case, the differences in size of partitions are even more exaggerated. For this company, division 1 is all of the United States, and represents greater than 85% of the entire business. But there are a number of divisions in other countries. The retailer currently replicates the data model for each country in order to provide reasonably efficient access to the data for an individual country. However, this is difficult to maintain, and limits the ability to do analyses across the entire company.

With multilevel partitioning, all the data can be stored in the same table, and analyses can be performed efficiently across either all the data, or with subsets of the data. This example is also applicable to manufacturers with multiple divisions and date associated data.

Importance of Partition Order for Specifying Partitioning Expressions

The specification order of partitioning expressions can be important for multilevel partitioning. The system maps multilevel partitioning expressions into a single-level combined partitioning expression. It then maps the resulting combined partition number 1-to-1 to an internal partition number. Rows are in logical RowID order, where a RowID consists of an internal partition number, a row hash value, and a row uniqueness value (RowIDs are identical for single-level and multilevel partitioning).

This implies a partial physical ordering based on how the file system manages the data blocks and cylinders, though this is not a strict relationship. This physical ordering maintains the ordering of partitions (except for a possible wraparound of internal numbers at one point in the internal number sequence for each level). Multilevel partitioning expressions are analogous with a single-level partitioning expression that is identical to the combined partitioning expression for multilevel partitioning, at least in terms of expressing how rows are logically ordered by the file system.

There are several implications of this ordering.

- Query performance

Row partition elimination at the lowest levels can increase overhead because of the frequent need to skip to the next internal partition to be read. This is because a partition at a lower level is split among the partitions at higher levels in the partition hierarchy.

At higher levels in the partition hierarchy, there are more contiguous internal partitions to scan and skip.

- Bulk data load performance

If a load is order-based on one of the partitioning columns, having that partitioning column at the highest level in the partition hierarchy can improve load performance because those partitions are contiguous. For example, a date-based partition with daily data loads might benefit from having the date-based partitioning at the first level.

You define the order of the partitioning expressions, and that ordering implies the logically ordering by RowID. Because the partitions at each level are distributed among the partitions of the next higher level in the hierarchy, scanning a partition at a certain level requires skipping some internal partitions.

If the number of rows in each internal partition is large, then skipping to the next internal partition to be read incurs relatively little overhead.

If the system reads or skips only a few rows for each internal partition, performance might be worse than a full-table scan.

Partition expression order does not affect the ability to eliminate partitions, but does affect the efficiency of a partition scan. As a general rule, this should not be a concern if there are many rows or, more specifically, multiple datablocks in each of the nonempty internal partitions.

Consider the following 2 cases.

- For a table with 65,535 combined partitions, the maximum number of combined partitions per partitioned primary index is 6.5535×10^9 (billion) rows per AMP, 100 byte rows, and 50 KB data blocks, and assuming an equal distribution of rows among the partitions, each combined partition spans 200 to 201 data blocks.

In this case, skipping over internal partitions should not incur significant overhead. Either all, or nearly all, of the rows in the data blocks read qualify, and the system skips at least 199 data blocks between each set of data blocks read.

- If the table has only 6.5535×10^6 (million) rows per AMP, each combined partition has about 0.2 data blocks. In the worst case, where the system eliminates only every other partition at the lowest level, every data block is read, and a full-table scan would be more efficient.

If there are 5 partitions at the lowest level, and the system can eliminate 4 out of 5 partitions, it still must read every data block.

If there are 6 partitions at the lowest level, and the system can eliminate 5 out of the 6, it can skip some data blocks, but possibly not enough to overcome the overhead burden, so this might not be more efficient than a full-table scan.

With a large number of partitions at the lowest level, and a large number of eliminated partitions at the lowest level, the system can probably skip enough more data blocks that the overhead burden can be overcome, and the operation might be more efficient than a full-table scan.

This second case is somewhat artificial, and probably not a good use of multilevel partitioning. Instead of the demonstrated case, you should consider an alternative partitioning that results in multiple data blocks for each nonempty internal partition.

A partitioning scheme that defines fewer levels of partitioning and fewer partitions per level, where the lowest level has the greatest number of partitions, ensures that there are more rows per combined partition, and would be far more useful. For example, if one level was initially partitioned in intervals of one day, changing the interval to one week or one month might be better.

Row partition elimination at the lower partition levels of a row partition hierarchy requires more skipping, which can both cause more I/O operations and increase the CPU path length.

To achieve optimal performance, you should specify a partitioning expression that is more likely to evoke row partition elimination for queries at a higher level and specify those expressions that are not as likely to evoke row partition elimination either at a lower level, or not at all. You should also consider specifying the row partitioning expression with the greatest number of row partitions at the lowest level.

As previously noted, the order of the row partitioning expressions might not be a significant concern if there are many data blocks per nonempty combined partition.

You can use the following query to find the average number of rows per populated combined row partition.

```
SELECT AVG(pc)
FROM (SELECT COUNT(*) AS pc
      FROM t
      GROUP BY PARTITION) AS pt;
```

Assuming the average block size is b and the row size is r , you can use the following query to find the average number of data blocks per nonempty combined row partition. The ideal number of data blocks per nonempty combined row partition is 10 or more.

```
USING (b FLOAT, r FLOAT)
SELECT (:r / :b) * AVG(pc)
FROM (SELECT COUNT(*) AS pc
      FROM t
      GROUP BY PARTITION) AS pt;
```

Different Multilevel Row Partitioning of the Same Table

The purpose of this example is to demonstrate properties of different partitionings of the same data.

Stage 1: First multilevel row partitioning of the orders table.

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey1 INTEGER,
  o_custkey2 INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey1 BETWEEN 0
                        AND      50
                        EACH    10), /* p1 */
              RANGE_N(o_custkey2 BETWEEN 0
                        AND      100
                        EACH    10)) /* p2 */;
```

This definition implies the following information about the row partitioning of orders.

- Number of row partitions in the first level, or highest, level = 6
- Number of row partitions in second, or lowest, level = 11
- Total number of combined partitions = 66
- Combined partitioning expression = $(p1 - 1) * d2 + p2 = (p1 - 1) * 11 + p2$

Now if o_custkey1 is 15 and o_custkey2 is 55, the following additional information is implied:

- Row partition number for level 1 = 2.
- Row partition number for level 2 = 6.
- PARTITION#L3 through PARTITION#L62 are all 0.
- Combined partition number = PARTITION = $(2-1)*11 + 6 = 17$.

The following case examples demonstrate some of the properties of this partitioning scheme.

```
SELECT * FROM orders WHERE o_custkey1 = 15;
```

In this case, there might be qualifying rows in the row partitions where the combined row partition numbers = $(2-1)*11 + (1 \text{ to } 11) = 11 + (1 \text{ to } 11) = 12 \text{ to } 22$.

```
SELECT * FROM orders WHERE (o_custkey1 = 15) OR o_custkey1 = 25 AND
o_custkey 2 BETWEEN 20 AND 50;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in (14 to 17, 25 to 28).

```
SELECT * FROM ORDERS WHERE o_custkey2 BETWEEN 42 AND 47;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in (5, 16, 27, 38, 49, 60).

Stage 2:

Suppose you submit the following ALTER TABLE request on orders.

```
ALTER TABLE orders
MODIFY PRIMARY INDEX
  DROP RANGE BETWEEN 0
                AND 9
                EACH 10
  ADD RANGE BETWEEN 51
                AND 70
                EACH 10,
  DROP RANGE BETWEEN 100
                AND 100
  ADD RANGE -100 TO -2;
```

This alters the row partitioning expressions to be the following expressions.

```
RANGE_N(o_custkey1 BETWEEN 10
                AND 50
                EACH 10, 51 AND 70
                EACH 10),
RANGE_N(o_custkey2 BETWEEN -100
                AND -2, 0
                AND 99
                EACH 10)
```

In other words, the table definition after you have performed the ALTER TABLE request is as follows.

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey1 INTEGER,
  o_custkey2 INTEGER)
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey1 BETWEEN 10
                        AND 50
                        EACH 10, /* p1 */
                        51
                        AND 70
```

```

      EACH      10),
RANGE_N(o_custkey2 BETWEEN -100
      AND      -2, 0
      AND      99
      EACH      10)) /* p2 */;

```

This changes the information implied by the initial table definition about the row partitioning of orders as follows.

- Number of partitions in the first, and highest, level = $d1 = 7$
- Number of partitions in the second, and lowest, level = $d2 = 11$
- Total number of combined partitions = $d1 * d2 = 7 * 11 = 77$
- Combined partitioning expression = $(p1 - 1) * d2 + p2 = (p1 - 1) * 11 + p2$

Now if `o_custkey1` is 15 and `o_custkey2` is 55, the following additional information is implied.

- Partition number for level 1 = `PARTITION#L1` = $p1(15) = 1$.
- Partition number for level 2 = `PARTITION#L2` = $p2(55) = 7$.
- `PARTITION#L3` through `PARTITION#L15` are all 0.
- Combined partition number = `PARTITION` = $(1-1)*11 + 7 = 7$.

The following cases provide examples of how multilevel row partitioning might be useful.

```
SELECT * FROM orders WHERE o_custkey1 = 15;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in the range $(1-1)*11 + (1 \text{ TO } 11) = 1 \text{ TO } 11$.

```
SELECT * FROM orders WHERE (o_custkey1 = 15 OR o_custkey1 = 25)
AND o_custkey2 BETWEEN 20 AND 50;
```

In this case, there might be qualifying rows in the row partitions where the combined partition numbers are in the ranges $4 \text{ TO } 7$ and $15 \text{ TO } 18$.

```
SELECT * FROM orders WHERE o_custkey2 BETWEEN 42 AND 47;
```

In this case, there might be qualifying rows in the row partitions where the combined partition number is any of the following.

- 6
- 17
- 28
- 39
- 50
- 61

- 72

Three-Level Row Partitioning Example

The following CREATE TABLE request defines a table with three levels of row partitioning.

```
CREATE TABLE sales (
  storeid      INTEGER NOT NULL,
  productid    INTEGER NOT NULL,
  salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  totalrevenue DECIMAL(13,2),
  totalsold    INTEGER,
  note         VARCHAR(256))
UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
PARTITION BY (RANGE_N(salesdate BETWEEN DATE '2003-01-01'
                        AND      DATE '2005-12-31'
                        EACH INTERVAL '1' YEAR),
              RANGE_N(storeid  BETWEEN 1
                        AND      300
                        EACH 100),
              RANGE_N(productid BETWEEN 1
                        AND      400
                        EACH 100));
```

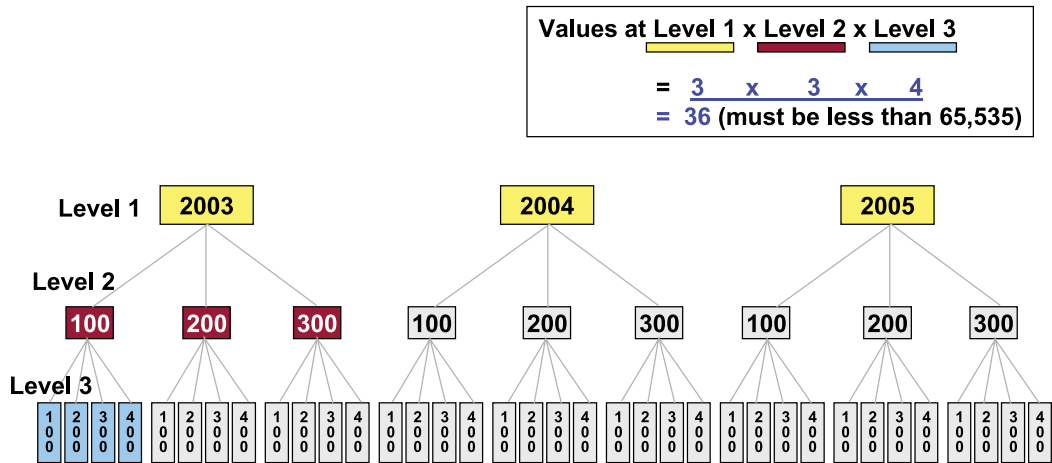
Combined Partitioning Expression for the Sales Table

The corresponding combined partitioning expression is the following.

```
(RANGE_N(salesdate  BETWEEN DATE '2003-01-01'
                        AND      DATE '2005-12-31'
                        EACH INTERVAL '1' YEAR)-1)*12+
(RANGE_N(storeid    BETWEEN 1
                        AND      300
                        EACH 100)-1)*4+
(RANGE_N(productid  BETWEEN 1
                        AND      400
                        EACH 100)
```

There are 3 row partitions for both the first level and second levels, and 4 row partitions for the third level. Therefore, the total number of combined partitions is the product of $3 \times 3 \times 4$, or 36.

The following diagram indicates the logical hierarchy of the three row partitioning levels for the sales table (65,535 assumes 2-byte partitioning):



How Rows for the Sales Table Are Grouped on an AMP

The following table shows how rows would be grouped on an AMP and, for each row, the partition number for each level (the result of the row partitioning expression for that level) and combined partition number (the result of the combined partitioning expression).

Rows are grouped by the combined partition number, and within a group are ordered by hash value first, then by uniqueness value. The Note column is truncated in all of these examples.

Note that for this example, only one sample row is shown for each combined partition number.

You can see that all the rows for a particular year are grouped together in this table; therefore, accessing those rows can be accomplished by reading only a subset of the data on each of the AMPs.

9: Primary Index, Primary AMP Index, and NoPI Objects

Combined Partition Number	Partition Number			Sales					
	L1	L2	L3	Storeid	Productid	Salesdate	Total Revenue	Total Sold	Note
1	1	1	1	96	10	2003-04-15	4158	42	Good day
2	1	1	2	71	184	2003-07-06	1972	68	Marginal Sa
3	1	1	3	80	241	2003-11-09	3055	47	Slow day
4	1	1	4	82	363	2003-12-24	1261	13	Promotion
5	1	2	1	186	1	2003-01-11	255	17	Shelf Life
6	1	2	2	122	163	2003-06-13	2405	65	Multiple
7	1	2	3	110	234	2003-05-01	2618	77	2 for 1
8	1	2	4	187	384	2003-08-03	684	9	Buy 3, 1 Fr
9	1	3	1	280	31	2003-03-10	493	29	Rain
10	1	3	2	202	116	2003-02-17	6732	68	Cash Sale
11	1	3	3	292	272	2003-01-30	439	11	Quarterly
12	1	3	4	213	385	2003-04-29	4233	83	Promotion
13	2	1	1	76	51	2004-10-05	442	34	2 for 1
14	2	1	2	26	149	2004-01-09	365	5	Good day
15	2	1	3	57	295	2004-11-04	2684	61	Marginal Sa
16	2	1	4	87	338	2004-08-02	696	58	Slow day
17	2	2	1	108	34	2004-04-27	4218	74	2 for 1
18	2	2	2	171	143	2004-10-30	5925	79	Shelf Life
19	2	2	3	114	228	2004-05-24	2064	48	Rain

9: Primary Index, Primary AMP Index, and NoPI Objects

20	2	2	4	135	347	2004-08-03	110	55	Promotion
21	2	3	1	257	75	2004-09-18	3417	51	Promotion
22	2	3	2	295	191	2004-11-11	376	8	Quarterly
23	2	3	3	208	204	2004-03-17	864	36	Multiple
24	2	3	4	221	330	2004-12-15	1456	52	Buy 3, 1 Fr
25	3	1	1	39	85	2005-09-15	192	48	Cash Sale
26	3	1	2	55	112	2005-07-12	232	29	Shelf Life
27	3	1	3	76	243	2005-03-13	6696	93	2 for 1
28	3	1	4	7	309	2005-08-20	116	58	Discounted
29	3	2	1	186	44	2005-05-30	4275	75	Credit
30	3	2	2	183	167	2005-06-14	2982	42	Promotion
31	3	2	3	171	218	2005-05-22	80	8	Rain
32	3	2	4	180	389	2005-06-09	4128	96	Good day
33	3	3	1	278	61	2005-09-22	1581	51	Buy 4, 1 Fr
34	3	3	2	256	110	2005-04-22	3280	80	Sale
35	3	3	3	246	233	2005-11-29	5133	59	Discounted
36	3	3	4	251	342	2005-02-11	968	44	50% off

For example:

- All the rows with a storeid in the range of 101 through 200, inclusively, can be found in the row partitions for combined partition numbers 5 to 8, 17 to 20, and 29 to 32.

As a result, if you were to specify a predicate of `WHERE storeid BETWEEN 101 AND 200`, Vantage can use row partition elimination to scan only combined partitions 5-8, 17-20, and 29-32 because only they contain rows that evaluate to TRUE for the condition.

- All the rows with a productid in the range of 201 through 300, inclusively, can be found in the combined partition numbers 3, 7, 11, 15, 19, 23, 27, 31, and 35.

As a result, if you were to specify a predicate of `WHERE productid BETWEEN 201 AND 300`, Vantage can use row partition elimination to scan only partitions 3, 7, 11, 15, 19, 23, 27, 31, and 35 because only they contain rows that evaluate to TRUE for the condition.

- All the rows with a storeid in the range of 1 and 100, inclusively, and a productid in the range 301 through 400, inclusively, can be found in the combined partitions for combined partition numbers 4, 16, and 28.

As a result, if you were to specify a predicate of `WHERE storeid BETWEEN 1 AND 100 AND productid BETWEEN 301 AND 400`, Vantage can use row partition elimination to scan only combined partitions 4, 16, and 28 because only they contain rows that evaluate to TRUE for the condition.

Similarly, Vantage can find rows by reading only a subset of the data for other combinations of two or more conditions on the partitioning columns. If you are looking for rows with a specific value of each of the partitioning columns, then only one combined partition for the specific combined row partition number needs to be read.

Performance Implications of Multilevel Row Partitioning

- If a SELECT request specifies values for all the primary index columns, Vantage can determine the AMP on which the rows reside, and only a single AMP needs to be accessed.

If conditions are not specified on the partitioning columns, then Vantage can probe each combined partition to find the rows based on their hash value.

If conditions are also specified on the partitioning columns, row partition elimination might reduce the number of row partitions to be probed on that AMP.

- If a SELECT request does not specify the values for all the primary index columns or there is no primary index, then Vantage must do an all-AMP full-table scan for a nonpartitioned table.

However, if row partitioning is defined on the table, and if you specify conditions on the partitioning columns, row partition elimination can reduce an all-AMP full file scan to an all-AMP scan of only the partitions of the combined partitioning expression that are not eliminated.

The degree of row partition elimination that can be achieved depends on the partitioning expressions, the conditions in the query, and the ability of the Optimizer to recognize such opportunities.

You need not specify values for all the partitioning columns in a query for row partition elimination to occur. Row partition elimination occurs at each level independently; the combination of the row partition elimination, if any, for each level determines which combined partitions need to be processed.

Summary of Primary Index Selection Criteria

The following table summarizes the guidelines for selecting columns to be used as primary indexes.

Guideline	Comments
Select columns that are most frequently used to access rows.	Restrict selection to columns that are either unique or highly singular.
Select columns that are most frequently used in equality predicate conditions.	Equality conditions permit the system to hash directly to the row having the conditional value. When the primary index is unique, the response is never more than one row. Inequality conditions require additional processing.
Select columns that distribute rows evenly across the AMPs.	Distinct values distribute evenly across all AMPs in the configuration. This maximizes parallel processing. Rows having duplicate NUPI values hash to the same AMP and often are stored in the same data block. This is good when rows are only moderately nonunique. Rows having NUPI columns that are highly nonunique distribute unevenly, use multiple data blocks, and incur multiple I/Os. Extremely nonunique primary index values can skew space usage so markedly that the system returns a message indicating that the database is full even when it is not. This occurs when an AMP exceeds the maximum bytes threshold for a user or database calculated by dividing the PERMANENT = n BYTES specification by the number of AMPs in the configuration, causing the system to incorrectly perceive the database to be "full."
Select columns that are not volatile.	Volatile columns force frequent row redistribution.
Select columns having very many more distinct values than the number of AMPs in the configuration.	If this guideline is not followed, row distribution skews heavily, not only wasting disk space, but also devastating system performance. This rule is particularly important for large tables.
Do not select columns defined with Period, ARRAY, VARRAY, Geospatial, JSON, XML, BLOB, CLOB, XML-based UDT, BLOB-based UDT, or CLOB-based UDT data types.	You cannot specify columns that have BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, ARRAY, VARRAY, Geospatial, or JSON data types in a primary index definition. If you attempt to do so, the CREATE request aborts. You can, however, specify Period data type columns in the partitioning expression of a partitioned table.
Do not select aggregated columns of a join index.	When defining the primary index for a join index, you cannot specify any aggregated columns. If you attempt to do so, the CREATE JOIN INDEX request aborts.

Principal Criteria for Selecting a Primary Index

When assigning columns to be the primary index for a table, there are three essential factors to keep in mind: uniform distribution of rows, optimal access to the data, and the volatility of indexed column values.

You will sometimes encounter situations where the selection criteria conflict. For example, specifying a NUPI instead of a UPI, or specifying an alternate key as the UPI instead of the primary key.

There are additional criteria to evaluate when selecting the primary index for a queue table. See [Selecting a Primary Index for a Queue Table](#) for a description of the primary index selection criteria you need to evaluate when choosing a primary index for a queue table.

Keep in mind that these criteria apply only to selecting a column set for the primary index. They do not apply to making a decision whether the primary index should be row-partitioned or not.

Be aware that with the exception of column-partitioned tables, Vantage assigns a default primary index to a table if you do not specify an explicit PRIMARY INDEX or NO PRIMARY INDEX in the CREATE TABLE request you use to create the definition for the table (see [Primary Index Defaults](#)).

Uniform Data Distribution

With respect to uniform data distribution, you should always consider the following factors.

- The more distinct the primary index values, the better.
- Rows having the same primary index value are distributed to the same AMP.
- Parallel processing is more efficient when table rows are distributed evenly across the AMPs.

Optimal Data Access

With respect to optimal data access using a primary index, you should consider the following factors.

- The primary index should be chosen on the most frequently used access path.

For example, if rows are generally accessed by a range query, you should consider defining a partitioned primary index on the table or join index that creates a useful set of partitions.

If the table is frequently joined with a specific set of tables, then you should consider defining the primary index on the column set that is typically used as the join condition.

- Primary index operations must provide the full primary index value.
- Primary index retrievals on a single value are always single-AMP operations.

Index Column Volatility

The primary index column set should be rarely, and preferably never, updated.

Criteria for Selecting a Primary Index

The following guidelines and performance considerations apply to selecting a unique or a nonunique column set as the primary index for a table.

- Choose columns for the primary index based on the selection set most frequently used to retrieve rows from the table even when that set is not unique (if and only if the values of the selection set are fairly equally distributed across the AMPs).
- Choose columns for the primary index that do not have XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, JSON, ARRAY, VARRAY, or VARIANT_TYPE data types. Distinct and structured UDT columns are valid components of a primary index, but UDT columns based on internal Teradata UDT types, such as the Period, Geospatial, ARRAY, and VARRAY types, are not.
- Choose columns for the primary index that distribute table rows evenly across the AMPs. The more singular the values for a column, the more optimal their distribution.
- Choose as few columns as possible for the primary index to optimize its generality.

All the columns in a composite primary index must be specified in a WHERE clause predicate before the Optimizer can select it for use as the retrieval mechanism.

- If it is difficult to define a unique primary index for a table that must have one, you can generate arbitrary unique values for a single column if you define it as an identity column with the characteristics ALWAYS GENERATED and NO CYCLE.
- Base the column selection on an equality search (if the primary index is a PPI, then the search is done within each non-eliminated populated partition). For equality constraints only, the system hashes directly to the row set that satisfies the condition.

Tables with this kind of primary key ...	Tend to assign the primary index to ...
single-column	the primary key. This is referred to as a Unique Primary Index (UPI).
multicolumn	one of the foreign key components of the primary key. This is referred to as a Nonunique Primary Index (NUPI).

- Primary and other alternate key column sets often can provide useful uniqueness constraints as well as a powerful access and join method when the logical design for a table is physically realized. If the primary or other alternate keys for a table are not selected to be its primary index, you should consider assigning a unique constraint, such as PRIMARY INDEX, UNIQUE, or a USI on those keys if the uniqueness constraint would facilitate table access and joins.

This recommendation is contingent on a number of complicated factors that must be considered before implementing unique constraints. See [Using Unique Secondary Indexes to Enforce Row Uniqueness](#) for a list of the factors that should be considered when you consider implementing this recommendation.

A UPI ...	WHILE a NUPI ...
at most involves one row	can involve multiple rows.
does not require a spool	often creates a spool.

- Duplicate NUPI values are always stored on the same AMP and in the same data block if possible.

- NUPI retrieval only requires one I/O operation (or two I/Os if the cylinder index is not memory-resident) when the rows are stored in the same data block.

This type of value range ...	Seen when using this predicate in a WHERE clause ...	Results in this kind of retrieval action ...
implicit	BETWEEN	full table scan, irrespective of any indexes defined for the table. The exceptions are the following: <ul style="list-style-type: none"> ◦ PPI tables and join indexes, where row partition elimination can be exploited. ◦ Hash and join index tables with a value-ordered NUPI, where value ordering can be exploited.
explicit	IN	individual row hashing.

Considerations for Choosing a Primary Index

Selecting the optimum primary index for a table or uncompressed join index is often a complex task because some applications might favor one type of primary index, while other applications might perform more optimally using a different primary index. Tables can have only one primary index, however, so you must select one that best suits the majority of the applications that a table serves. Of course, if the overhead costs justify the expense, you can define multiple join indexes with different primary indexes.

You can always add additional indexes, such as secondary, hash, and join indexes, to facilitate particular applications. Be aware that these indexes all incur various overhead costs, including:

- Disk space required to store their subtables.
- System performance degrades whenever base table rows are updated because the index values for any indexed columns affected by that update must also be updated.

You should always consider these tradeoffs when planning your indexes, then be sure to test them to ensure that the assumptions that lead to your choices are correct. For example, if you design a primary index with even row distribution as your principal criterion, analyze the actual distribution of table rows to ensure that they are evenly distributed.

For many applications, particularly those that use range queries heavily, a partitioned primary index can provide a better solution to resolving these issues than a nonpartitioned primary index because it provides efficient access both via the primary index columns as well as via a constraint on the partitioning columns. As always, you should confirm that the partitioning actually improves query performance by carefully examining EXPLAIN reports and collecting the appropriate statistics.

You should always collect statistics on the PARTITION column and the partitioning columns.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You should also weigh the costs of the index against the benefits it provides. This is particularly important if you have also defined a USI on the table because additional maintenance is required to enforce uniqueness, thus potentially neutralizing or even reducing the overall performance advantage of the index.

Creating a partitioned table does not guarantee that row-partition elimination plan. A partitioning might not be used for any of the following common reasons:

- It is not applicable to the actual queries in the workload.
- The Optimizer cost analysis for a query determines that another plan is less expensive.
- The query does not conform to any number of restrictions.

In some cases, a query plan with partitioning might not perform as well as one without partitioning.

Various partitioning strategies can be followed.

- For some applications, defining the partition expressions such that each row partition has approximately the same number of rows might be an effective strategy.

This task is far easier for single-level PPIs than for multilevel PPIs, though it can still be thought of as a goal to be approximated as best as possible.

- For other applications, having a varying number of rows per partition might be desirable. For example, more frequently accessed data (such as for the current year) might be divided into finer partitions (such as weeks) but other data (such as previous years) may have coarser partitions (such as months or multiples of months).

Note that partitioning in this manner can make altering the partitions more difficult.

- Alternatively, defining each range with equal width, even if the number of rows per range varies, might be important.

The most important factors for row partitioning are accessibility and maximization of row partition elimination. In all cases, defining a primary index (or having no primary index) that distributes the rows of the table fairly evenly across the AMPs is critical for efficient parallel processing.

Partitioning Guidelines

The following guidelines provide a high-level set of criteria for making an initial evaluation of whether row partitioning would provide more benefits to a query workload than a nonpartitioned table.

- Large tables and join indexes are usually better candidates for row partitioning than smaller tables and join indexes because there is not much benefit to partitioning a table or join index small enough that a full-table scan on the nonpartitioned table or join index takes only a few seconds.

The exception to this is a small table that is row-partitioned identically to a larger table with which it is frequently joined and with which it shares its primary index.

- When possible, you should row-partition on sets of columns that are frequently used as query conditions. For example, if half the queries against a table specify a date range that qualifies less than 25% of the rows, then that date column is a good candidate to be the partitioning column for the table.

If there is no column that is frequently used as a query condition, then there is probably little or no advantage to row-partitioning the table.

- All factors being equal, it is better to partition on a column set that is part of the primary index column set than to partition on a column that is not.

The exception to this is if the primary index is rarely, if ever, used for row access or join operations.

- Keep the number of row partitions relatively small. The key word in this guideline is relatively. The guideline also applies for multilevel partitioning situations, though it is more difficult to achieve for multilevel partitioning because the total number of partitions is a multiplicative factor of the number of partitioning levels defined for the table, so the number of partitions can grow very quickly even when there are few partitions defined for each level.

The exception to this guideline is if the primary index is rarely, if ever, used for row access or direct merge joins.

For example, if all the queries against the table access at least one month of activity, there is little or no benefit to partitioning by week or day instead of by month. An exception to this is if bulk data loading times are greatly reduced by a finer partition granularity.

See [Scenario 4](#) for an example of evaluating these sorts of tradeoffs.

- Keep the number of partitions small even if you plan to expand predetermined table operations in the future. You can always increase the number of partitions later when they are needed.

Note:

If you collect and maintain fresh statistics on the PARTITION columns of tables, this consideration is much less important.

You have greater flexibility with this guideline for single-level partitioned tables than you do for multilevel partitioned tables because it can be rather complicated to decrease the number of partitions for a multilevel partitioning because the number of combined partitions defined for such a table increases multiplicatively with each partition and with each level defined.

-
- The same criteria for selecting the column set for a nonpartitioned table also apply to partitioned tables.

Choose a primary index column set that provides good row distribution, avoids skew, and is commonly used to access individual rows or do not use a primary index.

Optimal row distribution and frequent access are sometimes conflicting considerations, so you must evaluate their relative merits and come to some compromise if that is the case.

Evaluating the Relative Merits of Partitioning Versus Not Partitioning

The following criteria provide a high-level means for evaluating the relative merits of partitioning or not partitioning for a table.

Potential advantages of row partitioning.

- The greatest potential gain in row-partitioning a primary index is the ability to read a subset of table or join index rows instead of scanning them all.

For example, a query that examines two months of sales data from a table that maintains two years of sales history can read about 1/12 of the table instead of having to scan it all.

The advantages of row partition elimination can be even greater for multilevel partitioned tables (see the examples of static partition elimination in *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for some remarkable scan optimizations).

This provides the opportunity for a large performance boost to a wide range of queries. Importantly, the individuals who code those queries do not have to know the partitioning structure of the table and, as a result, there is no need to recode existing SQL applications.

- Appropriate row partitioning can also facilitate faster batch data loads.

For example, if a table is partitioned by transaction date, the loading of transactions for the current day can be dramatically enhanced, as can the deletion of rows from the table that are no longer necessary.

- Row partitioning can make one or more existing secondary, hash, or join indexes redundant, which permits them to be dropped from the database.

Potential disadvantages of row partitioning.

- Row partitioning can make single row (primary index) accesses to the table slower if a partitioning column is not a member of the primary index column set.

This disadvantage can be offset somewhat by using one of the following strategies:

- Choose a partitioning column that is a member of the primary index column set.
- Define a unique secondary index that can be used to make single row accesses to the table.
- Constrain the values of the partitioning column set to enable the Optimizer to eliminate some row partitions when the query search conditions permit.
- Row partitioning can make direct merge joins of tables slower unless both tables are partitioned identically.

This disadvantage can be offset when query search conditions allow some row partitions to be excluded from the join operation.

The Teradata query optimizer has several special product join and merge join methods available to it just for joining row-partitioned tables. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for descriptions and examples of these join methods.

As with other physical database design choices, you must always evaluate the respective tradeoffs of the decisions that are available to you by prototyping and testing their relative merits.

Selecting a Primary Index for a Queue Table

Selection of a primary index might or might not be important when defining a queue table (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about creating and using queue tables). If you do not define a primary index explicitly, the system uses the QITS (Queue Insertion Time Stamp) column, the first column in the queue table, as the default. The values of the QITS column reflect the time of insertion of their rows. If you need to perform a primary index update or

delete one row from a queue table, you can, in most cases, easily browse the table and get the value of the QITS and other columns needed for the single-AMP update activity.

However, you might also choose a primary index column set based on the input data that is easily known, to support frequent updating. The queue table definition presented below has *orderkey* as its primary index, for example.

```
CREATE TABLE event02_QT, QUEUE (
  table_event02_QT_QITS TIMESTAMP(6) NOT NULL
                                DEFAULT CURRENT_TIMESTAMP(6),
  orderkey                    DECIMAL(18,0) NOT NULL,
  productkey                  DECIMAL(18,0) NOT NULL)
PRIMARY INDEX (orderkey);
```

Using a business entity for the primary index makes sense if you have a requirement to reorder the queue on a regular basis (or otherwise manipulate the rows), and the number of rows it holds is not trivial, making browsing the entire queue table for each update less desirable.

When you perform INSERT ... SELECT processing from a staging table into a queue table, you must pay closer attention to the selection of the primary index for the queue table. Such an INSERT ... SELECT operation might perform a similar function as a trigger when doing row-at-a-time insert operations: select the few rows that compose events and insert them immediately into a queue table for further processing.

If you are using a minibatch approach to loading data, then using set processing to identify events makes sense. In this case, the WHERE clause for the INSERT ... SELECT statement contains the event-identification criteria. In the case illustrated by the previous figure, having a queue table with the same primary index definition as the staging table improves the efficiency of the INSERT ... SELECT processing. The two inserts into the staging and queue tables then occur on the same AMP, eliminating the overhead of row redistribution.

Designing a queue table to share the same primary index as another base table could also be useful for tactical queries (see [Design Issues for Tactical Queries](#)). A given tactical application might want either to peek into the queue or to seek out the presence of a specific row in the queue. If the primary index value of the other table is known, and it is also the primary index value of a possible queue table row, then the system enables single-AMP access.

A similar situation exists if Teradata Parallel Data Pump inserts are made into a target table that has a trigger into a queue table. If the queue table and the target table share the same primary index, and you want to serialize the input to avoid cross-session blocking, serializing on the base table primary index also causes inserts into the queue table to be serialized effectively.

Column Distribution Demographics and Primary Index Selection

Effects of Skew on Query Processing

A skewed rows per value measure does not necessarily indicate a problem. It is often possible to have an even distribution of rows across AMPs and evenly distributed workloads when executing a query against skewed data. As long as the Optimizer has good statistics to work with, it is quite good at generating good query plans even when the distribution of table rows is skewed.

It is always better if the data is not skewed, but Vantage is equipped to deal with skew, and can often process skewed data successfully. Sometimes data is just naturally skewed, and the Optimizer has no choice but to deal with it.

SQL Scripts For Detecting Skew

Notice that several different measures of skew can be made.

- Rows per value
- Rows per hash bucket
- Rows per AMP
- Rows per row partition
- Rows per row partition per AMP
- Rows per row partition per hash bucket
- Rows per row partition per value

Each type of skew can have a different effect on the query plan the Optimizer chooses, as does the concentration of the relevant rows within the data blocks.

If you are analyzing the demographics of an existing table, whether production or prototype, you can use the following set of useful scripts written by the Teradata technical support team to check for data skew. You can adjust the details of these statement to suit the needs of your site.

This is a good practice to undertake when new applications are being loaded on the system. It is also good practice to run these queries regularly if there are many data changes.

The following query identifies tables that are not evenly distributed. Ideally, variance should be less than 5%. In this query, variance is set to 1000%, which generally indicates that some or many AMPs have no rows from the table in question. You can use the BTEQ command RETLIMIT to limit the number of rows returned.

```
SELECT (MAX(CurrentPerm) - MIN(CurrentPerm)) * 100
      /(NULLIF(MIN(currentperm),0))(NAMED variance)
      (FORMAT 'zzzzz9.99%'),MAX(CurrentPerm)(TITLE 'Max')
      (FORMAT 'zzz,zzz,zzz,999'),MIN(currentperm)
      (TITLE 'Min')(FORMAT 'zzz,zzz,zzz,999'),
      TRIM(DatabaseName)||'.'||TableName (NAMED Tables)
FROM DBC.TablesizeV
GROUP BY DatabaseName, TableName
```

```
HAVING SUM(CurrentPerm) > 1000000
AND    variance > 1000
WHERE DatabaseName NOT IN('CrashDumps','DBC')
ORDER BY Tables;
```

Use the following query to display the detailed distribution of a table that has been identified as having a skewed distribution:

```
SELECT vproc, CurrentPerm
FROM DBC.TablesizeV
WHERE DatabaseName = '<databasename>'
AND    TableName = '<tablename>'
ORDER BY 1;
```

The following query reports the row distribution of a table by AMP:

```
SELECT dt1.a (TITLE 'AMP'), dt1.b (TITLE 'Rows'),
      ((dt1.b/dt2.x (FLOAT)) - 1.0)*100 (FORMAT'+++9%',
      TITLE 'Deviation')
FROM (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>))),COUNT(*)
      FROM <databasename>.<tablename>
      GROUP BY 1) AS dt1 (a,b),
      (SELECT (COUNT(*) / (HASHAMP()+1)(FLOAT))
      FROM <databasename>.<tablename>) AS dt2(x)
ORDER BY 2 DESC,1;
```

The following query reports the distribution by AMP of the specified index or column.

```
SELECT HASHAMP(HASHBUCKET(HASHROW(<index or column>))) ,COUNT(*)
FROM <databasename>.<tablename>
GROUP BY 1
ORDER BY 2 DESC;
```

The following query reports the number of row hash collisions for the specified index or column.

```
SELECT HASHROW(index or column), COUNT(*)
FROM <databasename>.<tablename>
GROUP BY 1
ORDER BY 1
HAVING COUNT(*) > 10;
```

The following query reports the number of AMPs and the number of rows a given query accesses.

```
LOCKING TABLE <tablename> FOR ACCESS
SELECT COUNT(dt.ampNum)(TITLE '#AMPS'),
       SUM(dt.numRows)(TITLE '#ROWS')
FROM (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>))), COUNT(*)
      FROM <tablename>
      WHERE <selection criteria>
      GROUP BY 1)AS dt (ampNum, numRows);
```

Scenario 1

Table Structure and Update Schedule

Assume that a retail enterprise has a large, nonpartitioned primary index sales table containing the details of each transaction for the previous 24 full months plus the current month-to-date. Once a month, the transactions from the oldest month are deleted from the table. Current transactions are loaded into the table nightly using Teradata Parallel Transporter. Most transactions are added on the date they occur, but a small percentage of transactions might be reported a few days after they occur. The number of transactions per month is roughly the same for all months.

Each row contains, among other things, the product code for the item, the transaction date, an identifier for the sales agent, and the quantity sold. The rows are short, and the data blocks are large. The primary index is a composite of product code, transaction date, and the agent identifier.

Query Workload

The query workload is a mix of tactical and strategic requests:

- There is a modest volume of short-running single-AMP (primary index) queries.
- There are many ad hoc queries follow a general pattern of comparing current-month-to-date sales to the same days of the previous month, or to the same days of the same month of the previous year for a few product code values.
- Some queries analyze agent performance, usually over an interval of a calendar quarter or less.
- Some queries examine sales trends over the previous 24 full months, usually for most or all product code values.

All the tables that support the workload have different primary indexes.

The sales table is frequently joined to relatively small tables containing information about each product code and each sales agent.

Problem Statement

The current definition of *sales_table* does not use row partitioning.

```
CREATE TABLE sale_stable (
  product_code      CHARACTER(8),
  sales_date        DATE,
```

```

agent_id          CHARACTER(8),
quantity_sold     INTEGER,
product_description VARCHAR(50))
PRIMARY INDEX (product_code,sales_date,agent_id);

```

The DBA has been told to speed up the ad hoc queries and agent analysis queries. He considers two possible optimizations, neither of which uses row partitioning.

- Define either a UPI or a join index on the transaction date column.

The DBA then sets up tests for both scenarios. Unfortunately, the EXPLAIN reports show that the optimizer finds neither index to be selective enough to improve performance over a full-table scan, and does not use them.

- Split the table into 25 separate tables, each containing transactions for a calendar month, and then define a view that UNIONS all the tables. This view is intended to be used by the applications that analyze 24 months of sales history.

After some analysis, he concludes that this solution could indeed speed up the targeted queries, but that it also adds too much complexity for his end users, who would now have to understand the view structure and change the table names in their queries, code more complicated UNION statements, and select appropriate date and product code ranges. The requirement to know the right table name also applies to the short-running single-AMP queries that specify primary index values. This proposed solution also complicates the nightly load jobs, especially in the first few days of a month when a small number of the transactions would be from the prior month, as well as complicating his long successful archive strategy. The DBA ultimately rejects this alternative as being too complicated and error-prone.

With these negative results in hand, the DBA next considers redefining the sales table with row partitioning.

Analysis of Row Partitioning Benefits

Thinking that row partitioning might be an easy way to optimize his workloads for this situation, the DBA converts the sales table into a partitioned table partitioned by transaction month and finds that many of his critical queries run faster with no significant negative tradeoffs.

The DDL for the redefined sales table looks like this:

```

CREATE TABLE ppi_sales_table (
  product_code      CHARACTER(8),
  sales_date        DATE,
  agent_id          CHARACTER(8),
  quantity_sold     INTEGER,
  product_description VARCHAR(50))
PRIMARY INDEX (product_code,sales_date,agent_id)
PARTITION BY RANGE_N(sales_date BETWEEN DATE '2001-06-01'
                     AND      DATE '2003-06-31'
                     EACH INTERVAL '1' MONTH);

```


The following description examines each element of the workload as it relates to the newly redefined sales table, *ppi_sales_table*.

- The monthly deletes are faster because instead of a monthly Teradata Parallel Transporter delete job, the DBA can now perform a simple monthly ALTER TABLE statement to do the following things:
 1. Drop the entire partition that contains the oldest data.
 2. Create a few new partitions to hold data for the next few future months.

Deleting all the rows in a partition is optimized in much the same way that deleting all the rows in a nonpartitioned primary index table is optimized. For example.

- There is no need to record the individual rows in the transient journal as they are deleted.
- The rows for the month being deleted are stored contiguously on each AMP instead of being scattered more or less evenly among all the data blocks of the table, so there are fewer blocks to read.
- Most of the deletes are full-block deletes, so the block does not need to be rewritten.
- There is no need to touch any of the rows for the other months.

Dropping the oldest partition set is a nearly instantaneous operation, assuming there are no USIs or join indexes that must be updated.

The DDL for this ALTER TABLE statement looks like this for a selected monthly update:

```
ALTER TABLE sales_table
MODIFY PRIMARY INDEX (product_code, sales_date, agent_id)
DROP RANGE BETWEEN DATE '2001-06-01'
                AND    DATE '2001-06-30'
ADD RANGE BETWEEN DATE '2003-07-01'
                AND    DATE '2003-07-31'
WITH DELETE;
```

- The nightly Teradata Parallel Transporter insert job should run somewhat faster than before. Instead of the inserted rows distributing more or less evenly among all the data blocks of the table, they are concentrated in data blocks that correspond to the proper month. This increases the average hits per block count, which is a key measure of Teradata Parallel Transporter efficiency, as well as reducing the number of blocks that must be rewritten.
- The short-running, single-AMP queries are not affected by partitioning. Because the partitioning column is a component of the primary index, primary index access to the table is not changed.
- The largest gain is seen in the ad hoc queries comparing current month sales to a prior month. Only two of the 25 partitions need to be read, instead of the full-table scan required for a nonpartitioned table. This means that the number of disk reads will be reduced by roughly 92%, with a corresponding reduction in elapsed query response time. The 92% figure applies to the step that reads the sales table, not to the sum of all the steps used to implement the query.

Given the stated assumptions, the other steps should take roughly the same amount of time to complete as they did previously.

- The same considerations apply to the agent analysis queries. The number of partitions that must be read is determined by the time period specified in the query is known in most cases to be three or fewer. Even if the analysis is over twelve full months, there is roughly a 50% gain in reading twelve of 25 partitions for the step that reads the sales table.
- The decision support queries that analyze 24 months of sales data take roughly the same time and resources as they did for a nonpartitioned primary index table, although there is a marginal gain realized by reading 24 instead of 25 partitions. If the analysis is for 24 months plus the current month (the entire table), then the resource usage is the same as with the nonpartitioned primary index incarnation of the table because full-table scans are not affected by partitioning the primary index.
- The joins take the same amount of time. In the defined workload, the EXPLAIN reports indicate that there are no direct joins against the sales table. Instead, spool is always created from the sales table and the spool is then joined to the smaller tables. In the partitioned table case, the spool might be created more efficiently, but once created it is exactly the same as it is for the nonpartitioned table.

Conclusions

The results show there are few disadvantages, and some significant advantages, to row partitioning this table given the workloads that access it. The partitioned table requires somewhat more disk space than its nonpartitioned counterpart. There is a 2-byte or 8-byte partition number recorded in each row that consumes additional storage space; however, the percentage increase seen for most row sizes does not exceed about 5%, and is often considerably less than that.

The following table summarizes the improvement opportunities for this case study:

Activity	Nonpartitioned Table	Partitioned Table	Improvement	Comments
Monthly delete of one month of data	Teradata Parallel Transporter job reads most blocks, updates most blocks	ALTER TABLE statement deletes partition	Much faster performance	Easier maintenance
Nightly inserts	Inserted rows scattered throughout table	Inserted rows concentrated in one partition	Faster performance	No changes to load script needed
Primary index access	1 block read	1 block read	No change	No SQL changes needed
Comparison of current month to prior month	All blocks read	2 partitions read	Step is 12 times faster (only 2 of 25 partitions read)	No SQL changes needed
Trend analysis over entire table	All blocks read	All blocks read	Little change	<ul style="list-style-type: none"> • Rows are two bytes longer for partitioning • 2% more blocks for 100 byte rows
Joins	No direct Merge Joins	No direct Merge Joins	Little change	No direct merge joins in this

Activity	Nonpartitioned Table	Partitioned Table	Improvement	Comments
				example because of the choice of primary index.

Scenario 2

Table Structure and Update Schedule, Query Workload, and Problem Statement

This case study is a continuation of the scenario presented in [Scenario 1](#). The table structure, update schedule, query workload, and problem statement are unchanged. The scenario evaluates whether the partitioning definition for this table can be improved further.

Analysis of Row Partitioning Benefits

The partitioning definition used for [Scenario 1](#) partitioned by month because many of the queries use months as their basic time unit. Another option is to attempt to optimize access further by changing the granularity of the partitioning scheme by partitioning at a finer level. Suppose the DBA considers partitioning the table by day instead of by month. The table now has about 760 partitions (25 months), with those corresponding to future dates in the current month being empty.

The results of telescoping the partition granularity for this table and query workload look like this:

- The effort to delete the oldest month of data is substantially the same as it was for the same table defined with 25 primary index partitions. The run time for that job is also substantially the same. Making the simplifying assumption that all months have 30 days, the DBA would have to delete 30 smaller partitions with the oldest month of data instead of deleting one larger partition, but the same number of rows are deleted, and the elapsed time to delete them is roughly the same.
- The nightly inserts benefit from the finer partitioning because instead of being concentrated in one or two partitions out of 25, the inserted rows are now directed to three, four, or at most five partitions out of 760; well under 1% of the partitions. Most of the inserts are directed to one partition, the one containing the activity for the day just completed.
- The short-running primary index access queries are not impacted either way by having 760 partitions instead of 25.
- The ad hoc queries that usually analyze two or three months of data are also little changed, now accessing roughly 60 out of 760 partitions rather than 2 out of 25, roughly the same percentage of the defined partitions for the primary index. Queries varying by the time of month, however, might realize some performance gain with the larger number of partitions. For example, a query submitted on the fourth day of the month might be likely to analyze the current day and the previous 33 days of data, while a query submitted later in the month might restrict the analysis to that calendar month. In this case, the 34 day query would involve 68 out of 760 partitions, instead of 4 out of 25, a significantly smaller percentage.

- The analysis queries examining 24 months of data run in about the same length of time because they touch most of the rows in the table in either case.
- The joins are not impacted by the number of partitions because there are no direct joins against this table.

Conclusions

Increasing the number of partitions from 25 to 760 has only a modest effect on performance for this particular workload. The greatest gain is for queries that analyze only a few days of transactions.

Scenario 3

Table Structure and Update Schedule

While a transaction date or timestamp is frequently a good choice for the partitioning column, other choices present themselves for consideration for other categories of workloads and data. Consider a table with detailed information about telephone calls maintained by a telecommunications company. This table stores rows that contain the originating telephone number, a timestamp for the beginning of the call, and the duration of the call, among other things, for each outgoing call. Rows are retained for a variable length of time, but rarely for more than six weeks. Retention is based on the call date and the monthly bill preparation date. The primary index is a composite of the telephone number and the call-start timestamp. This column set implies that the index was chosen to provide good distribution, not to facilitate data access, and also that the likelihood of any direct primary index joins is remote.

Query Workload

Some queries analyze all calls from a particular telephone number, while others analyze all calls for a particular period of time, perhaps for as long as a month, for customers meeting certain criteria.

Problem Statement

The current definition of the call detail table does not use a partitioned primary index:

```
CREATE TABLE calldetail (
  phone_number      DECIMAL(10) NOT NULL,
  call_start        TIMESTAMP,
  call_duration     INTEGER,
  call_description  VARCHAR(30))
PRIMARY INDEX (phone_number, call_start);
```

Can the standard query workloads against this table be optimized by partitioning its primary index?

Analysis of Partitioning Benefits

There are two candidate schemes for partitioning this table. The first partitioning scheme is to partition the rows by *call_start*, probably defining one partition for each day. The results of this partitioning scheme for this table and query workload look like this.

- The timestamp partitioning scheme helps with inserting new transaction activity in the same way as the previous scenario ([Scenario 1](#)).
- Little performance gain is realized for row deletion because the delete operations are not performed strictly by call date. Because of this, the deleted rows are not clustered in a single partition.
- The analysis queries based on *call_start* benefit from this partitioning scheme, with those queries that specify a range of a few days realizing the greatest gain.

The second possibility for partitioning this table is to use the telephone number. Telephone numbers contain too many digits to assign each number its own partition, but a subset of the digits can be used profitably to optimize telephone number-based row access. If, for example, the table is partitioned on the high order three digits of the telephone number, there are 1,000 partitions, some of which are always be empty because of the way telephone companies assign numbers. The results of this partitioning scheme for this table and query workload look like this:

- This partitioning scheme does not improve the performance of Teradata Parallel Transporter bulk inserts or deletes because they are scattered across all partitions.
- The scheme does not facilitate date-based queries.
- The scheme allows queries that specify a telephone number to run much faster than they otherwise would because only one partition out of 500 or more partitions must be read to access the rows having that number.
- The scheme benefits geographic area analysis, at least in North America, because the first three digits of North American telephone numbers uniquely identify a narrowly defined geographic region.

If 1,000 partitions improve performance, then defining 10,000 partitions using the first four digits of the telephone number would probably improve performance even more. If 10,000 partitions are good, then 50,000 partitions might be better yet.

The DDL for the redefined *call_detail* table looks like this:

```
CREATE TABLE ppi_call_detail (
  phone_number      DECIMAL(10) NOT NULL,
  call_start        TIMESTAMP,
  call_duration     INTEGER,
  call_description  VARCHAR(30))
PRIMARY INDEX (phone_number, call_start)
PARTITION BY RANGE_N(phone_number/100000 (INTEGER) BETWEEN 0
                                AND 99999
                                EACH      2);
```

If mapping a geographic area to one or more partitions does not solve an application problem, then another potential solution is to maximize the number of partitions by using telephone number modulo 65,535 as the partitioning expression. Assuming the cardinality of the table is roughly 3.276 billion rows, then the average partition contains roughly 50,000 rows with this scheme. If the system has 100 AMPS, then each AMP contains roughly 500 rows per partition, a number that fits into one data block if the row width is fairly narrow.

The decrease in response time of a single-partition scan for all activity for a particular telephone number is dramatic compared to the full-table scan that would be required for a nonpartitioned table.

A query to return activity for one telephone number from this table is a best case scenario for single-table response time improvement by using row partitioning. Disregarding the overhead cost of initiating the query and returning the answer set, the elapsed time could be reduced to 1/65535 of the time using a nonpartitioned table. Including the query initiation and termination overhead, the total query time improvement would be somewhat less than a factor of 65,535, but could be less than 1/1,000 of the nonpartitioned primary index time. The DDL for the *ppi_call_detail* table using this scheme is as follows.

```
CREATE TABLE ppi_call_detail (
  phone_number DECIMAL(10)NOT NULL,
  call_start   TIMESTAMP,
  call_duration INTEGER,
  other_columns CHARACTER(30))
PRIMARY INDEX (phone_number, call_start)
PARTITION BY phone_number MOD 65535 +1;
```

The workload mix determines which, if any, of these proposed partitioning schemes is best. You can begin your analysis with the extended logical data model, but actual testing of the various proposed scenarios is often required.

Conclusions

The partitioning scheme you use should be tied directly to the application workload accessing its data. For this scenario, the proposed partitioning schemes affect geographic localization applications differently than they do call date-based applications. To decide which scheme is better, you need to know all the various ways the table is currently accessed and have a solid concept of how it might be accessed in the future.

You need to examine the relative efficiencies of different partitioning schemes for the same table must from that perspective.

Scenario 4

Dealing With Ambiguous Scenarios

The previous scenarios illustrate situations where a row-partitioned table is the obvious choice to enhance the performance of a query workload. This scenario examines a more ambiguous situation in which there are more tradeoff considerations and it is not possible to determine in advance one correct solution for all specific instances of the scenario.

Table Structure and Update Schedule

An invoice table contains data about each invoice issued in the past four years. The unique primary index is invoice number. New rows are added nightly, using Teradata Parallel Transporter Update Operator, and the oldest month of data is deleted once each month.

Query Workload

A moderately heavy volume of queries requests information about one specified invoice. There are also ad hoc analysis queries that examine all invoices for some period of time, which is usually less than one year. Other tables have invoice number as their primary index, but do not have an invoice date column, so there are frequent joins with those other tables.

Problem Statement

The DBA is considering whether it would be advantageous to partition the invoice table on invoice date using one-month ranges.

The primary index is currently defined as unique, but would have to be redefined as nonunique if the table were row-partitioned. There is a business requirement to guarantee that invoice numbers are unique, so the DBA would need to define a uniqueness constraint on the invoice number column. If this uniqueness constraint is added, it creates an additional secondary index on the table (other than UPIs, all uniqueness constraints are implemented internally as USIs irrespective of whether they are specified explicitly as a UNIQUE constraint, a PRIMARY KEY constraint, or a USI constraint. See [Using Unique Secondary Indexes to Enforce Row Uniqueness](#)), which increases processing on insert, delete, and update operations, as well as requiring additional disk capacity to store the resulting secondary index subtable. The base table is also larger by two bytes per row, further increasing the required disk space.

Analysis of Partitioning Benefits

The primary index access queries that were run against the nonpartitioned version of this table must be reformulated to use the USI to access the row. As a general rule, accessing a row takes roughly two to three times longer using a USI than it would using a UPI. However, UPI access is a very fast operation, so doubling or tripling the time might barely be noticeable to the users who issue those queries.

Without row partition elimination, direct Merge Joins require, at best, more memory and CPU utilization and might be measurably slower compared to a similar nonpartitioned table. The extent of performance degradation depends on the query conditions, how many partitions can be excluded, and the specific join plan chosen by the Optimizer. Actual measurement of representative queries is necessary to determine the overall difference in performance.

The nightly inserts benefit in the same way, and for the same reasons, as in [Scenario 1](#). However, the additional index on invoice number partially offsets that benefit. The same considerations apply to the monthly delete operations.

The ad hoc queries examining several months of invoices benefit in the same way as in [Scenario 1](#). The benefit is greatest when fewer months are examined.

Would it be worthwhile to convert the invoice table to a partitioned table? The DBA must measure the degree of improvement as well as the extent of degradation in the various types of query, and use that analysis to determine how much each query type contributes to the overall workload involving this table. This exercise produces a good estimate of the comparative workload performance against the table with and without partitioning.

If the measured performance difference between the otherwise equivalent partitioned and nonpartitioned tables is substantial, in either direction, then the choice might appear to be obvious. However, you must also weigh the relative importance to the enterprise of the various activities in the workload.

For example, consider the following contingencies:

- If the time required to perform the nightly volume of bulk inserts is beginning to exceed the time allotted for inserting new rows, then even a small improvement in load time might be considered sufficiently important to offset larger degradations in other aspects of the query workload.
- Similarly, if the response time of the PI-access queries is critical, even a small performance degradation might be considered unacceptable, whether net workload performance is improved or not.

Conclusions

The decision whether to implement a table with or without partitioning is not always cut and dried, and the ultimate decision, like many others in physical database design, can often be more of an optimization than a maximization. In this scenario, careful and considered measurement, analysis, and evaluation are all required to make an optimal decision.

Performance Considerations for Primary Indexes

This topic describes general performance considerations for both unique and nonunique primary indexes, paying particular attention to how the range of primary index singularity affects performance.

Guideline for All Primary Indexes

Define the primary index for a table on as few columns as possible. Retrievals on columns that do not match the entire primary index do not use the primary index to retrieve the matching rows.

Hashing efficiency increases as the number of primary index columns decreases.

Guidelines for Row Partitioning

The key guideline for determining the optimum granularity for the row partitions of a partitioned table is the nature of the workloads that most commonly access the PPI table or join index. The higher the number of row partitions you define for a partitioned table, the more likely an appropriate range query against the table will perform more quickly, given that the partition granularity is such that the Optimizer can eliminate all but one partition.

On the other hand, it is generally best to avoid specifying too fine a partition granularity. For example if query workloads never access data at a granularity of less than one month, there is no benefit to be gained by defining partitions with a granularity of less than one month. Furthermore, unnecessarily fine partition granularity is likely to increase the maintenance load for a partitioned table, which can lead to overall system performance degradation. So even though too fine a partition granularity itself does not introduce performance degradations, the underlying maintenance on such a table can indirectly degrade performance.

You should also consider the following items if a table you are designing is planned to support tactical queries. By knowing the specifics of your workloads, you can optimize your partitioning to best suit the queries in those workloads.

- For all-AMP tactical queries against partitioned tables, you should specify a constraint on the partitioning column set in the WHERE clause.
- If a query joins partitioned tables that are partitioned identically, using their common partitioning column set as a constraint enhances join performance still more if you also include an equality constraint between the partitioning columns of the joined tables.

General Considerations

The following factors apply equally to UPIs and NUPIs.

- Each table has 0 or 1, and no more than 1, primary index.
- The primary index for a table need not be isomorphic with the primary key for the table. This is obviously true for NUPIs, because primary key values are unique by definition.
- A primary index can contain nulls, while a primary key cannot.

Primary index nulls are not a good thing. If you anticipate that the primary index for a table might frequently be null, you should consider using a different column.

- All single-value primary index accesses are confined to a single AMP. In most cases, a single-value primary index access requires only 1 I/O.

If the cylinder index is not in cache, a primary index access requires 2 I/Os.

Unique Primary Index Considerations

The following factors describe the general performance considerations associated with UPIs.

- UPIs guarantee an even distribution of table rows across the AMPs when there are many more unique values than AMPs on the system.
- UPIs always guarantee the best retrieval and update performance for single table row access.
- UPI values are never associated more than one base table row.
- UPIs retrievals and updates never require spool.
- Uniqueness is enforced by the system using the UPI (thereby avoiding the need for duplicate row checks for multiset tables).

Nonunique Primary Index Considerations

The following factors describe the general performance considerations associated with NUPIs.

- At best, NUPIs distribute table rows evenly across the AMPs and hash values.

At worst, NUPI table row distribution can be skewed in various ways (see [SQL Scripts for Detecting Skew](#)) and, if so, decrement both retrieval and update performance.

- NUPIs at best can effect good retrieval and update performance.
- NUPI values can involve more than one base table row.

- NUPI retrievals can require spool.
- Duplicate NUPI values hash to the same AMP and often are stored in the same data block.
- If all the rows for a duplicate NUPI hash value fit into a single data block, then only 1 or 2 I/Os are required to store or access the entire set.
2 I/Os are only required when the cylinder index is not cached.
- If duplicate rows are excluded because the table is defined as SET and has no uniqueness constraint, then the system must make a duplicate row check for every table row inserted or updated.

See [Duplicate Row Checks for SET Tables with NUPIs](#) and [Using Unique Secondary Indexes to Enforce Row Uniqueness](#) for further details.

Duplicate Row Checks for SET Tables with NUPIs

Database tables defined with the SET attribute in the CREATE TABLE kind clause do not permit duplicate rows. Any time a row is inserted into a SET table having a NUPI and no uniqueness constraints or indexes, the system performs a duplicate row check.

When Vantage checks for duplicate rows, it considers nulls to be equal, while in SQL conditions, null comparisons evaluate to UNKNOWN. See [Inconsistencies in How SQL Treats Nulls](#) for information about how nulls are interpreted by commercially available relational database management systems.

Minimizing Duplicate NUPI Row Checks

There are several ways to minimize or even eliminate duplicate NUPI row checks while still preventing duplicate row from being inserted. This topic describes some of the ways to achieve that goal.

- Use a non-primary index column set to define a UNIQUE constraint or USI.
- Use multiple columns to define NUPIs in order to make the index as close to being unique as possible.
- Keep the number of NUPI duplicate rows for each value below 100.

Secondary Indexes

This section describes Unique Secondary Indexes (USIs) and Nonunique Secondary Indexes (NUSIs).

When column demographics suggest their usefulness, the Optimizer selects secondary indexes to provide faster single row or set selection, depending on whether the index is a USI or a NUSI, respectively.

A secondary index is never required for database tables, but they can often improve system performance, particularly in decision support environments.

While secondary indexes are useful for optimizing repetitive and standardized queries, Vantage is also highly optimized to perform full-table scans in parallel. Because of the strength of full-table scan optimization, there is little reason to be heavy-handed about assigning multiple secondary indexes to a table.

You can either create secondary indexes when you create the table or you can add them later using the CREATE INDEX statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144). Unlike primary indexes, secondary indexes can be dropped and created at will.

For more information about how Vantage accesses data using a unique secondary index, see [USI Access](#).

For more information about how Vantage accesses data using a nonunique secondary index, see [NUSI Access and Performance](#).

For design issues related to secondary index support for tactical queries, see [Two-AMP Operations: USI Access and Tactical Queries](#) and [NUSI Access and Tactical Queries](#).

As is true for other types of indexes, you should keep the statistics on your secondary indexes as current as possible.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. See “COLLECT STATISTICS in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for details on how to do this.

Space Considerations

Creating a secondary index causes the system to build a subtable to contain its index rows, thus adding another set of rows that requires updating each time a table row is inserted, deleted, or updated.

Secondary index subtables are also duplicated whenever a table is defined with FALLBACK.

When compression at the data block level is enabled for their primary table, secondary index subtables are not compressed. See [Compression Types Supported by Vantage](#) for more information about data block compression.

Unique Secondary Indexes

USIs are useful both for base table access (because USI access is, at worst, a two-AMP operation) and for enforcing data integrity by applying a uniqueness constraint on a column set. Like a unique primary index, a unique secondary index can be used to guarantee row uniqueness.

Using Unique Secondary Indexes to Enforce Row Uniqueness

When a non-primary index uniqueness constraint is created, whether it is a PRIMARY KEY or UNIQUE constraint, Vantage implements it as a USI.

As a general guideline, whenever you define a primary index for a multiset table to be a NUPI, particularly if the table is created in ANSI/ISO session mode (where the default for tables is multiset), you should consider defining one of the following uniqueness constraints on its primary key or other alternate key to facilitate row access and joins.

- Unique secondary index
- UNIQUE NOT NULL constraint
- PRIMARY KEY NOT NULL constraint

PRIMARY KEY and UNIQUE constraints are both mapped internally as USIs unless they are used to define the default UPI for a table. See [Primary Index Defaults](#).

USI Access

USI access is usually a two-AMP operation because Vantage typically distributes a USI row to a different AMP than the base table row the index points to. If the system distributes the USI subtable row to the same AMP as the base table row it points to, then only one AMP is accessed (but it is still a two-step operation).

The following stages are involved in a USI base table row access.

- The requested USI value is accessed by hashing to its subtable.
- The pointer to the base table row is read and used to access the stored row directly.

Unique Secondary Indexes and Performance

Statistics play an important part in optimizing access when USIs define conditions for the following operations.

- Joining tables
- Satisfying WHERE predicates that specify comparisons, string matching, or complex conditionals
- Satisfying LIKE expressions
- Processing aggregates

Because of the additional overhead for index maintenance, USI values should not change frequently. When you change the value of a secondary index, Vantage must undertake the following maintenance operations.

1. Delete secondary index references to the current value.
2. Generate secondary index references to the new value.

Creating a Unique Secondary Index as a Composite of a Row-Level Security Constraint Column and a NUPI Column Set

You can create a USI for a row-level security-protected table as a composite of a row-level security constraint column and the columns of a NUPI for the table. This property can be used to implement polyinstantiation.

Polyinstantiation is a property that allows a relation to contain multiple rows with the same primary key value, where the multiple instances are distinguished by their security levels, where a security level is defined by a row-level security constraint column.

Restrictions on Load Utilities

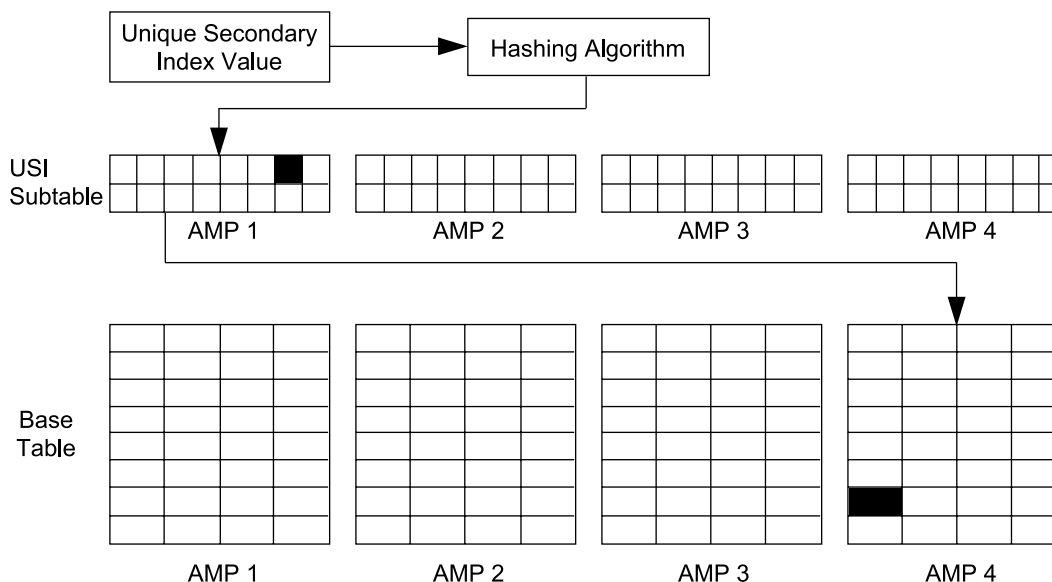
You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have unique secondary indexes.

Before you can load data into a USI-indexed base table, you must first drop all defined USIs before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, are supported for USI-indexed tables.

USI Hashing

USIs are hash-partitioned on their index columns, as indicated by the following graphic.



Nonunique Secondary Indexes

Nonunique secondary indexes are typically assigned to nonunique column sets that frequently appear in WHERE clause selection conditions, join conditions, ORDER BY and GROUP BY clauses, foreign keys, and miscellaneous other conditions such as UNION, DISTINCT, and any attribute that is frequently sorted.

Highly selective NUSIs are useful for reducing the cost of frequently made selections and joins on nonunique columns, and provide extremely fast access for equality conditions. This is particularly true for NoPI tables (see [NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes](#)), where the only other access method might be a full-table scan. Note that NUSIs with low selectivity can be less efficient than a full-table scan.

NUSIs are also useful for range access and in-list conditions and for geospatial indexes.

Also note the following about NUSIs:

- NUSI access is always an all-AMPs operation unless the index is defined on the same columns as the primary index. This is allowed when the NUSI is value-ordered or when the table or join index is partitioned and not all the partitioning columns are included in the primary index.
- The subtables must be scanned in order to locate the relevant pointers to base table rows. This is a fast lookup process when a NUSI is specified in an equality or range condition because the NUSI rows are either hash-ordered or value-ordered on each AMP.
- NUSI subtables are not covered by the active read fallback feature (see [Physical Database Integrity](#) for details).

Relationship Between a NUSI Subtable Row and Base Table Rows

A particular NUSI subtable row points to one or many base table rows on that same AMP. The relationship between a NUSI value and any individual AMP in a configuration is either 0:1, 1:1, or 1:M.

This relationship ...	Reflects the fact that ...
0:1	an AMP contains no NUSI subtable index rows for a particular NUSI value.
1:1	an AMPs contains 1 NUSI subtable index row for a particular NUSI value.
1:M	an AMP contains more than 1 NUSI subtable index row for a particular NUSI value.

NUSI Access and Performance

NUSI requests are all-AMP requests unless the NUSI is defined on the same columns as a primary index or primary AMP index.

The usefulness of a NUSI is correlated with the number of rows per value: the higher number of rows per value, the less useful the index. If the number of rows for a NUSI value exceeds the number of data blocks in the table, the usefulness of the NUSI might be questionable. On the other hand, as NUSI values approach uniqueness (meaning that the number of rows per value is either close to 1 or is significantly less than the number of AMPs in the system), an all-AMPs table access is wasteful and you should consider defining a join index (see [Join and Hash Indexes](#)) to support DML requests against the table instead of a NUSI.

Because NUSI access is usually an all-AMPs operation, NUSIs may seem to have limited value. If you have to access all AMPs in the configuration to locate the requested rows, why bother with an index?

The answer to that question is provided by the following list of ways that NUSIs can improve the performance of your decision support queries:

- NUSI access is often faster than a full-table scan, particularly for extremely large tables. A full-table scan is also an all-AMP operation.
- A NUSI that covers (see [NUSIs and Query Covering](#) for a definition of covering) the columns requested by a query is often included in Optimizer access plans.
- A NUSI that covers a LIKE expression or any selective inequality conditions is often included in Optimizer access plans.
- A NUSI on the same columns as the primary index (this is only allowed when the primary index does not include all the columns of all the partitioning columns) may be more efficient than accessing using the primary index when there is no or limited partition elimination for a query.

While NUSI access is usually an all-AMPs operation, keep in mind that the AMPs work in parallel. If all the AMPs have qualified rows, then this is a very efficient operation. If some or many of the AMPs do not have qualified rows, then those AMPs are doing work just to determine that they have no qualified rows. Note that if there are more rows per NUSI value than AMPs, it is likely that every AMP will have one or more qualified rows.

Depending on demographics and environmental cost variables, the Optimizer will specify a full-table scan instead of a NUSI access when it determines that the scan would be a more efficient access method.

Selectivity Considerations

Selectivity refers to the percentage of rows in a table containing the same nonunique secondary index value. An index that has high selectivity retrieves few rows. A unique primary index retrieval, for example, is highly selective because it never returns more than one row. An index that has low selectivity retrieves many rows.

Low Selectivity Indexes

When an index is said to have low selectivity, that means that many rows have the same NUSI value and there are relatively few distinct values.

A column with those characteristics is usually a poor choice for a NUSI because the cost of using it may be as high or higher than a full-table scan.

For example, assume that employee table contains 10,000 rows of about 100 bytes each, and there are only 10 different departments. If an average employee table data block is 2,560 bytes and can store about 25 rows, then the entire table requires about 400 data blocks.

If dept_no is defined as a nonunique secondary index on the employee table, and the dept_no values are evenly distributed, then the following query accesses about 1,000 row selections.

```
SELECT *
FROM employee
WHERE dept_no = 300;
```

Each AMP reads its own rows of the dept_no secondary index subtable. If any rows contain the index value 300, the AMP uses the associated rowIDs to select the data rows from its portion of the employee table.

Regardless of the number of AMPs involved, this retrieval requires 1,000 row selections from the employee table. To satisfy this number of select operations, it is likely that all 400 employee table data blocks would have to be read.

If that were the case, then the number of I/O operations undertaken by the retrieval could easily exceed the number required for a full-table scan. In such instances, a table scan would actually be a much more efficient solution than a NUSI-based retrieval.

High Selectivity Index

If deptno is a high selectivity index, where few employee rows share the same deptno value, then using deptno for retrieval provides better performance than a full-table scan. Because of these selective conditions, the Optimizer specifies NUSI access for request processing when it is less costly than a full-table scan.

NUSI Bit Mapping

Bit mapping is a technique used by the Optimizer to effectively link several low selectivity indexes in a way that creates a result that drastically reduces the number of base rows that must be accessed to retrieve the desired data. The process determines common rowIDs among multiple NUSI values by means of the logical intersection operation. The method is explained in more detail in the following sections.

Restrictions for When Bit Mapping Is Used

Vantage only performs NUSI bit mapping when low selectivity indexed conditions are ANDed but their composite selectivity is high. When WHERE conditions are connected by a logical OR, the Optimizer typically performs a full-table scan and applies the ORed conditions to every row without considering an index.

Note that this is not always the case. If low selectivity conditions ORed conditions are ANDed with other low selectivity conditions, but the composite selectivity is high, the Optimizer may choose to use a composite NUSI or a single-column NUSI.

Computing NUSI Bit Maps

We use the following query to show how NUSI bit maps are computed.

```
SELECT *
FROM employee_table
WHERE salary_amount > 20000
AND sex_code = 'M'
AND full_time = 'FT';
```

The following process illustrates the method for computing NUSI bit maps. The actions described occur concurrently on all AMPs.

1. The literals 20,000, M, and FT are used to access the corresponding rows from the index subtables.
2. For $n-1$ indexes, each AMP creates separate bit maps of all 0 values.

In this case, $n-1$ is 2 because there are three NUSIs defined, one for each condition in the WHERE clause.

3. The AMPs equate each base table rowID in each qualifying NUSI row with a single bit in the map, and each such map bit is turned ON.
4. The resulting bit maps are ANDed together to form the final bit map.
5. Each base table rowID in the remaining NUSI is equated to a single bit in the final bit map, and the state of that map is checked.

IF the map bit is ...	THEN ...
OFF (0)	no action is taken and the next map bit is checked.
ON (1)	the rowID is used to access the table row and all remaining, or residual, conditions are applied to that row.

Determining If Bit Mapping Is Being Used

To determine whether bit mapping is being used for your NUSIs, use the EXPLAIN request modifier and examine the reports it produces for your queries.

See the information about the EXPLAIN request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

NUSIs and Query Covering

The Optimizer aggressively pursues NUSIs when they can cover a query. *Covering* means that all of the columns requested in a query or data necessary for satisfying the query are also available from an existing index subtable, making it unnecessary to access the base table rows themselves to complete the query. Some vendors refer to this as index-only access.

Covering of a query can also be partial. In the case, the system can get the row IDs for those base table rows that possibly qualify for a query by preliminary examination from the index but then must also access the base table itself to retrieve the definitively qualified rows.

Example: Index Used to Cover a Query

The following example demonstrates a situation in which the Optimizer can use the index in place of the base table to satisfy the query.

```
CREATE INDEX idxord (o_orderkey, o_date, o_totalprice)
ON orders;
SELECT o_date, AVG(o_totalprice)
FROM orders
```

```
WHERE o_orderkey >1000
GROUP BY o_date;
```

Value-Ordered NUSIs and Range Conditions

Value-ordered NUSIs are very efficient for range conditions. Because the NUSI rows are sorted by data value, it is possible to search only a portion of the index subtable for a given range of key values.

Limitations

- The sort key is limited to a single numeric or DATE column.
- The sort key column cannot exceed four bytes in length.
- If defined over multiple columns and with an ORDER BY clause, they count as two consecutive indexes against the total of 32 non-primary indexes you can define on a base or join index table. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

One index represents the column list and the other index represents the ordering column.

Importance of Consecutive Indexes for Value-Ordered NUSIs

The system automatically assigns incrementally increasing numbers to indexes when they are created on a table. This is not important externally except for the case of composite value-ordered NUSIs, because these indexes not only consume two of the allotted 32 index numbers from the pool, but those two index numbers are consecutive. One index of the consecutively numbered pair represents the column list, while the other index in the pair represents the ordering column.

To understand why this is important, consider the following scenario.

1. You create 32 indexes on a table, none of which is value-ordered.

This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

2. You drop every other index on the table, meaning that you drop either all the odd-numbered indexes or all the even-numbered indexes.

For example, if you had dropped all the even-numbered indexes, there would now be 16 odd-numbered index numbers available to be assigned to indexes created in the future.

3. You attempt to create a value-ordered multicolumn NUSI.

The request aborts and the system returns an error message to you.

The reason the request aborts is that two consecutive index numbers were not available for assignment to the composite value-ordered NUSI.

You are still able to create 16 additional non-value-ordered NUSIs, single-column value-ordered NUSIs, USIs, hash indexes, or join indexes, but you cannot create any composite value-ordered NUSIs.

To work around this problem, perform the following procedure.

1. Drop any index on the table.

This action frees 2 consecutive index numbers.

2. Create the value-ordered multicolumn NUSI that failed previously.
3. Recreate the index you dropped to free the consecutive index numbers.

Here is a simple example of why value-ordered composite NUSIs consume two consecutive index numbers from the total available number of 32.

First define a table on which a NUSI is to be defined.

```
CREATE MULTISET TABLE transsupplier (
  supkey INTEGER NOT NULL,
  name    CHAR(25) CHARACTER SET LATIN CASESPECIFIC NOT NULL,
  seccode INTEGER NOT NULL,
  groupID INTEGER NOT NULL)
UNIQUE PRIMARY INDEX ( supkey );
```

Then you create an ordered composite NUSI on seccode and groupID, ordering on groupID.

```
CREATE INDEX (seccode, groupID)
ORDER BY VALUES (groupID) ON transsupplier;
```

You find two rows in DBC.Indexes with an IndexType code of V (meaning a value-ordered secondary index) and an IndexNumber of 4 for this composite NUSI. The third row, with an IndexType of I and an IndexNumber of 8, represents the fact that the groupID column is the ordering column of the two columns making up the composite NUSI, so a single ordered composite NUSI uses up two of the available 32 secondary, hash, or join indexes that can be defined for TransSupplier.

```
SELECT s.CreateTimestamp, s.IndexType, s.IndexNumber, t.TVMName,
       s.FieldName
FROM DBC.Indexes AS s, DBC.TVM AS t
WHERE s.tableID = t.TVMID
AND   s.databaseID = t.databaseID
AND   s.fieldID = s.fieldID
AND   s.tableID = s.tableID
AND   s.databaseID = s.databaseID
AND   s.indextype <> 'P'
AND   t.TVMName = 'Transsupplier'
ORDER BY s.createtimestamp DESC;
```

CreateTimeStamp	IndexType	IndexNumber	TVMName	FieldName
2007-04-27 14:51:55	I	8	TransSupplier	GROUPID

CreateTimeStamp	IndexType	IndexNumber	TVMName	FieldName
2007-04-27 14:51:55	V	4	TransSupplier	GROUPID
2007-04-27 14:51:55	V	4	TransSupplier	SECCODE

Suppose you had instead created the following ordered single-column NUSI on TransSupplier, this time on groupID only, ordering on groupID.

```
CREATE INDEX (groupID)
ORDER BY values (groupID) ON transsupplier;
```

You then run the same query against DBC.Indexes.

```
SELECT i.CreateTimestamp, i.IndexType, i.IndexNumber, t.TVMName,
       f.FieldName
FROM DBC.Indexes AS i, DBC.TVM AS t, DBC.TVFields AS f
WHERE i.tableID = t.TVMID
AND   i.databaseID = t.databaseID
AND   i.fieldID = f.fieldID
AND   i.tableID = f.tableID
AND   i.databaseID = f.databaseID
AND   i.indextype <> 'P'
AND   t.TVMName = 'Transsupplier'
ORDER BY i.createtimestamp DESC;
```

The query produces a report something like this, where you find only one row in DBC.Indexes with an IndexType code of V for this single-column NUSI, meaning that only one of the possible 32 secondary, hash, and join indexes has been consumed by the creation of this NUSI.

CreateTimeStamp	IndexType	IndexNumber	TVMName	FieldName
2007-04-27 14:56:24	V	4	TransSupplier	GROUPID

Typical Uses of Value-Ordered and Hash-Ordered NUSIs

The typical use of a hash-ordered NUSI is with an equality condition on the secondary index column set. The specified secondary key value is hashed, and then each NUSI subtable is probed for rows having the same row hash. For every matching NUSI entry, the corresponding rowIDs are used to access the base rows on the same AMP. Because the NUSI rows are stored in row hash order, searching the NUSI subtable for a particular row hash value is very efficient.

Value-ordered NUSIs, on the other hand, are useful for processing range conditions and conditions with either an equality or inequality on the ordering column of the secondary index.

The Optimizer can select hash-ordered NUSIs to access rows for range conditions using a full-table scan of the NUSI subtable to find the matching secondary key values.

With a value-ordered NUSI, it is possible to search only a portion of the index subtable for a given range of key values.

Example

The following example illustrates a value-ordered NUSI (defined by an ORDER BY clause that specifies the VALUES keyword option on o_orderdate) and a query that might be solved more efficiently if the specified value-ordered NUSI were selected by the Optimizer to access the requested rows.

```
CREATE INDEX Idx_Date (o_orderdate)
ORDER BY VALUES (o_orderdate)
ON Orders;
SELECT *
FROM Orders
WHERE o_orderdate
BETWEEN DATE '2005-10-01'
AND      DATE '2005-10-07';
```

Selecting a Secondary Index

When assigning columns to be a secondary index for a table, there are numerous factors to consider, the most important of all being the selectivity of the index.

While USI retrievals are always very efficient, the efficiency of NUSI retrievals varies greatly depending on their selectivity.

Optimal Data Access

Selectivity is a relative term that refers to the number of rows returned by an index. Most retrievals aim to return only a select few rows: very specific answers in response to a very specific request.

An index that returns a small number of rows is said to be highly selective. This is a positive attribute.

Indexes that return a large number of rows are said to have low selectivity. This is generally a negative attribute; so negative that, as often as not, the Optimizer selects a full-table scan over a NUSI with low selectivity because the full-table scan can be less costly.

All UPIs and USIs are highly selective by definition, as are most well-chosen NUPIs. High selectivity is favored not only because of its precision, but also because of its low cost, involving a very small number of disk I/Os, which is always a performance-enhancing attribute.

Criteria for Selecting a Secondary Index

The following rules of thumb and performance considerations apply to selecting a unique or nonunique column set as a secondary index for a table.

- Consider naming secondary indexes whenever possible using a standard naming convention.
- Avoid assigning secondary indexes to frequently updated column sets.
- Avoid assigning secondary indexes to columns with lumpy distributions because there is a slight chance the Optimizer might mistake their usefulness.
- Avoid creating excessive secondary indexes on a table, particularly for a table used heavily, or even moderately, for OLTP processing. The less frequently the table is updated, the more desirable a multiple index solution.
- Consider building secondary indexes on column sets frequently involved in the following clauses, predicates, and other logical operations:
 - Selection criteria
 - Join criteria
 - ORDER BY clauses
 - GROUP BY clauses
 - Foreign keys (because of join and subquery processing)
 - UNION, DISTINCT, and other sort operations

When these operations act on well-indexed column sets, the number of scans and sorts that must be performed on the data by the database manager can be greatly reduced.

- Consider creating USIs for tables without a UPI that require frequent single-row access.
- Consider creating NUSIs for tables that require frequent set selection access.
- Consider creating covering indexes when possible and cost effective (including considering the cost of maintaining the index). The Optimizer frequently selects covering indexes to substitute for a base table access whenever the overall cost of the query plan is reduced. Such index-only access promotes faster retrievals.

Alternatively, many applications are well served by join indexes, which can be used profitably in many covering situations where multiple columns are frequently joined. See [Join and Hash Indexes](#) for further information about join indexes.

- Consider creating secondary indexes on columns frequently operated on by built-in functions such as aggregates.
- Consider assigning a uniqueness constraint such as PRIMARY KEY, UNIQUE, or USI, as appropriate, to the primary or other alternate key of any table built with a NUPI. This both enforces uniqueness, eliminating the burden of making row uniqueness checks, and enhances retrieval for applications where the primary or other alternate key is frequently used as a selection or join criterion.

This guideline is situational and is contingent on a number of factors. The various factors involved in the recommendation are described in [Using Unique Secondary Indexes to Enforce Row Uniqueness](#).

A primary or alternate key USI might not be a good decision for a table that is frequently updated by OLTP applications.

- Plan to dynamically drop and recreate secondary indexes to accommodate specific processing and performance requirements such as bulk data loading utilities, database archives, and so on.

Create appropriate macros to perform these drop and create index operations if you need to undertake such specific processing tasks regularly.

- Make sure that your indexes are being used as planned by submitting EXPLAIN request modifiers to audit index selection for those queries they are designed to facilitate.

Secondary indexes that are never selected by the Optimizer are a burden to the system for the following reasons:

- They consume disk resources that could profitably be used to store data or indexes that are used.
- They degrade update processing performance unnecessarily.

Secondary Index Usage Summary

All secondary indexes have the following properties:

- Can enhance the speed of data retrieval.
- Do not affect base table data distribution.
- Maximum of 32 secondary, hash, and join indexes defined per table. Each composite NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation (see [Importance of Consecutive Indexes for Value-Ordered NUSIs](#)).

The limit of 32 indexes applies to any combination of secondary, hash, and join indexes defined on a table, ranging from 0 secondary indexes and 32 join indexes, 11 hash indexes, 11 join indexes, and 10 secondary indexes to 32 secondary indexes and 0 join indexes.

This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

- Can be composed of as many as 64 columns.
- Can include columns defined with a UDT data type.
- Cannot contain columns defined with XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, Period, or JSON data types.
- Cannot be defined on global temporary trace tables.
- Can be created or dropped dynamically as data usage changes or if they are found not to be useful for optimizing data retrieval performance.
- Require additional disk space to store subtables.
- Require additional I/Os on INSERTs, DELETEs, and possibly on UPDATEs and MERGEs.
- Should not be defined on columns whose values change frequently.
- Should not include columns that do not enhance selectivity.
- Should not use composite secondary indexes when multiple single-column indexes and bit mapping might be used instead.

- Composite secondary index is useful if it reduces the number of rows that must be accessed.
- Most efficient for selecting a small number of rows.
- NUSIs can be hash-ordered or value-ordered.
- Ordering for NUSIs defined with an ORDER BY clause is restricted to a single numeric or DATE column of 4 or fewer bytes.
- If they cover, or partially cover, a query, then they further improve their usefulness.

USI Summary

- Can be used to enforce row uniqueness for multiset NUPI and NoPI tables.
- Guarantee that each complete index value is unique.
- Any access is, at most, a two-AMP operation.

NUSI Summary

- Useful for locating rows having a specific value in the index.
- Can be hash-ordered or value-ordered.

Value-ordered NUSIs are particularly useful for enhancing the performance of range queries.

- Any access is an all-AMPs operation with the exception of the case where a NUSI is defined on the same column set as the primary index for the table.
- If an index is defined with an ORDER BY clause, it counts as 2 consecutive indexes against the table limit of 32 secondary, hash, and join indexes (see [Importance of Consecutive Indexes for Value-Ordered NUSIs](#)).

Related Information

For more information about secondary indexes, see the information about CREATE INDEX and CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Secondary Index Access Summarized by Example

This example indicates how a series of queries against a table can use various secondary indexes to access the rows in that table.

Configuration

The system for this example has four AMPs.

Table Definition

The table used in this demonstration is a simple three-column customer table, defined as follows.

Column Name	Attribute Described	Type of Index Defined on the Column
Cust	Customer Number	USI
Name	Customer Last Name	NUSI

Column Name	Attribute Described	Type of Index Defined on the Column
Phone	Customer Phone Number	NUPI

The following is a snapshot instance of this table.

customer table

Customer		
Cust	Name	Phone
USI	NUSI	NUPI
37	White	555-4444
98	Brown	333-9999
74	Smith	555-6666
95	Peters	555-7777
27	Jones	222-8888
56	Smith	555-7777
45	Adams	444-6666
31	Adams	111-2222
40	Smith	222-3333
72	Adams	666-7777
84	Rice	666-5555
49	Smith	111-6666
12	Young	777-7777
62	Black	444-5555
77	Jones	777-6666
51	Marsh	888-2222

Base table and secondary index subtable rows are distributed as illustrated by the following graphic.

<div>USI Subtable</div> <table><tr><th>RowID</th><th>Cust</th><th>RowID</th></tr><tr><td>244, 1</td><td>74</td><td>884, 1</td></tr><tr><td>505, 1</td><td>77</td><td>639, 1</td></tr><tr><td>744, 4</td><td>51</td><td>915, 9</td></tr><tr><td>757, 1</td><td>27</td><td>388, 1</td></tr></table>	RowID	Cust	RowID	244, 1	74	884, 1	505, 1	77	639, 1	744, 4	51	915, 9	757, 1	27	388, 1	<div>USI Subtable</div> <table><tr><th>RowID</th><th>Cust</th><th>RowID</th></tr><tr><td>135, 1</td><td>98</td><td>555, 6</td></tr><tr><td>296, 1</td><td>84</td><td>536, 5</td></tr><tr><td>602, 1</td><td>56</td><td>778, 7</td></tr><tr><td>969, 1</td><td>49</td><td>147, 1</td></tr></table>	RowID	Cust	RowID	135, 1	98	555, 6	296, 1	84	536, 5	602, 1	56	778, 7	969, 1	49	147, 1	<div>USI Subtable</div> <table><tr><th>RowID</th><th>Cust</th><th>RowID</th></tr><tr><td>288, 1</td><td>31</td><td>638, 1</td></tr><tr><td>339, 1</td><td>40</td><td>640, 1</td></tr><tr><td>372, 2</td><td>45</td><td>471, 1</td></tr><tr><td>588, 1</td><td>95</td><td>778, 3</td></tr></table>	RowID	Cust	RowID	288, 1	31	638, 1	339, 1	40	640, 1	372, 2	45	471, 1	588, 1	95	778, 3	<div>USI Subtable</div> <table><tr><th>RowID</th><th>Cust</th><th>RowID</th></tr><tr><td>175, 1</td><td>37</td><td>107, 1</td></tr><tr><td>489, 1</td><td>72</td><td>717, 2</td></tr><tr><td>838, 1</td><td>12</td><td>147, 2</td></tr><tr><td>919, 1</td><td>62</td><td>822, 1</td></tr></table>	RowID	Cust	RowID	175, 1	37	107, 1	489, 1	72	717, 2	838, 1	12	147, 2	919, 1	62	822, 1																																				
RowID	Cust	RowID																																																																																																	
244, 1	74	884, 1																																																																																																	
505, 1	77	639, 1																																																																																																	
744, 4	51	915, 9																																																																																																	
757, 1	27	388, 1																																																																																																	
RowID	Cust	RowID																																																																																																	
135, 1	98	555, 6																																																																																																	
296, 1	84	536, 5																																																																																																	
602, 1	56	778, 7																																																																																																	
969, 1	49	147, 1																																																																																																	
RowID	Cust	RowID																																																																																																	
288, 1	31	638, 1																																																																																																	
339, 1	40	640, 1																																																																																																	
372, 2	45	471, 1																																																																																																	
588, 1	95	778, 3																																																																																																	
RowID	Cust	RowID																																																																																																	
175, 1	37	107, 1																																																																																																	
489, 1	72	717, 2																																																																																																	
838, 1	12	147, 2																																																																																																	
919, 1	62	822, 1																																																																																																	
<div>NUSI Subtable</div> <table><tr><th>RowID</th><th>Name</th><th>RowID</th></tr><tr><td>448, 1</td><td>White</td><td>107, 1</td></tr><tr><td>656, 1</td><td>Rice</td><td>536, 5</td></tr><tr><td>567, 3</td><td>Adams</td><td>638, 1</td></tr><tr><td>432, 8</td><td>Smith</td><td>640, 1</td></tr></table>	RowID	Name	RowID	448, 1	White	107, 1	656, 1	Rice	536, 5	567, 3	Adams	638, 1	432, 8	Smith	640, 1	<div>NUSI Subtable</div> <table><tr><th>RowID</th><th>Name</th><th>RowID</th></tr><tr><td>567, 2</td><td>Adams</td><td>717, 2</td></tr><tr><td>372, 2</td><td>Adams</td><td>471, 1</td></tr><tr><td>852, 1</td><td>Brown</td><td>555, 6</td></tr><tr><td>432, 3</td><td>Smith</td><td>884, 1</td></tr></table>	RowID	Name	RowID	567, 2	Adams	717, 2	372, 2	Adams	471, 1	852, 1	Brown	555, 6	432, 3	Smith	884, 1	<div>NUSI Subtable</div> <table><tr><th>RowID</th><th>Name</th><th>RowID</th></tr><tr><td>432, 1</td><td>Smith</td><td>147, 1</td></tr><tr><td>770, 1</td><td>Young</td><td>147, 2</td></tr><tr><td>567, 6</td><td>Jones</td><td>338, 1</td></tr><tr><td>448, 4</td><td>Black</td><td>822, 1</td></tr></table>	RowID	Name	RowID	432, 1	Smith	147, 1	770, 1	Young	147, 2	567, 6	Jones	338, 1	448, 4	Black	822, 1	<div>NUSI Subtable</div> <table><tr><th>RowID</th><th>Name</th><th>RowID</th></tr><tr><td>262, 1</td><td>Jones</td><td>639, 1</td></tr><tr><td>396, 1</td><td>Peters</td><td>778, 3</td></tr><tr><td>432, 5</td><td>Smith</td><td>778, 7</td></tr><tr><td>155, 1</td><td>Marsh</td><td>915, 9</td></tr></table>	RowID	Name	RowID	262, 1	Jones	639, 1	396, 1	Peters	778, 3	432, 5	Smith	778, 7	155, 1	Marsh	915, 9																																				
RowID	Name	RowID																																																																																																	
448, 1	White	107, 1																																																																																																	
656, 1	Rice	536, 5																																																																																																	
567, 3	Adams	638, 1																																																																																																	
432, 8	Smith	640, 1																																																																																																	
RowID	Name	RowID																																																																																																	
567, 2	Adams	717, 2																																																																																																	
372, 2	Adams	471, 1																																																																																																	
852, 1	Brown	555, 6																																																																																																	
432, 3	Smith	884, 1																																																																																																	
RowID	Name	RowID																																																																																																	
432, 1	Smith	147, 1																																																																																																	
770, 1	Young	147, 2																																																																																																	
567, 6	Jones	338, 1																																																																																																	
448, 4	Black	822, 1																																																																																																	
RowID	Name	RowID																																																																																																	
262, 1	Jones	639, 1																																																																																																	
396, 1	Peters	778, 3																																																																																																	
432, 5	Smith	778, 7																																																																																																	
155, 1	Marsh	915, 9																																																																																																	
<div>Base Table</div> <table><tr><th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr><tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr><tr><td>107, 1</td><td>37</td><td>White</td><td>555-4444</td></tr><tr><td>536, 5</td><td>84</td><td>Rice</td><td>666-5555</td></tr><tr><td>638, 1</td><td>31</td><td>Adams</td><td>111-2222</td></tr><tr><td>640, 1</td><td>40</td><td>Smith</td><td>222-3333</td></tr></table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	107, 1	37	White	555-4444	536, 5	84	Rice	666-5555	638, 1	31	Adams	111-2222	640, 1	40	Smith	222-3333	<div>Base Table</div> <table><tr><th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr><tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr><tr><td>471, 1</td><td>45</td><td>Adams</td><td>444-6666</td></tr><tr><td>555, 6</td><td>98</td><td>Brown</td><td>333-9999</td></tr><tr><td>717, 2</td><td>72</td><td>Adams</td><td>666-7777</td></tr><tr><td>884, 1</td><td>74</td><td>Smith</td><td>555-6666</td></tr></table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	471, 1	45	Adams	444-6666	555, 6	98	Brown	333-9999	717, 2	72	Adams	666-7777	884, 1	74	Smith	555-6666	<div>Base Table</div> <table><tr><th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr><tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr><tr><td>147, 1</td><td>49</td><td>Smith</td><td>111-6666</td></tr><tr><td>147, 2</td><td>12</td><td>Young</td><td>777-4444</td></tr><tr><td>388, 1</td><td>27</td><td>Jones</td><td>222-8888</td></tr><tr><td>822, 1</td><td>62</td><td>Black</td><td>444-5555</td></tr></table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	147, 1	49	Smith	111-6666	147, 2	12	Young	777-4444	388, 1	27	Jones	222-8888	822, 1	62	Black	444-5555	<div>Base Table</div> <table><tr><th>RowID</th><th>Cust</th><th>Name</th><th>Phone</th></tr><tr><td></td><td>USI</td><td>NUSI</td><td>NUPI</td></tr><tr><td>639, 1</td><td>77</td><td>Jones</td><td>777-6666</td></tr><tr><td>778, 3</td><td>95</td><td>Peters</td><td>555-7777</td></tr><tr><td>778, 7</td><td>56</td><td>Smith</td><td>555-7777</td></tr><tr><td>915, 9</td><td>51</td><td>Marsh</td><td>888-2222</td></tr></table>	RowID	Cust	Name	Phone		USI	NUSI	NUPI	639, 1	77	Jones	777-6666	778, 3	95	Peters	555-7777	778, 7	56	Smith	555-7777	915, 9	51	Marsh	888-2222
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
107, 1	37	White	555-4444																																																																																																
536, 5	84	Rice	666-5555																																																																																																
638, 1	31	Adams	111-2222																																																																																																
640, 1	40	Smith	222-3333																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
471, 1	45	Adams	444-6666																																																																																																
555, 6	98	Brown	333-9999																																																																																																
717, 2	72	Adams	666-7777																																																																																																
884, 1	74	Smith	555-6666																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
147, 1	49	Smith	111-6666																																																																																																
147, 2	12	Young	777-4444																																																																																																
388, 1	27	Jones	222-8888																																																																																																
822, 1	62	Black	444-5555																																																																																																
RowID	Cust	Name	Phone																																																																																																
	USI	NUSI	NUPI																																																																																																
639, 1	77	Jones	777-6666																																																																																																
778, 3	95	Peters	555-7777																																																																																																
778, 7	56	Smith	555-7777																																																																																																
915, 9	51	Marsh	888-2222																																																																																																
AMP 1	AMP 2	AMP 3	AMP 4																																																																																																

Sample Queries

The following queries can all be answered without having to do a full-table scan. Note that *uniqueness value* is abbreviated UV throughout.

Query 1

The first query uses the NUPI column, phone, as the WHERE clause attribute, with the requested value being 555-7777.

```
SELECT *
FROM Customer
WHERE Phone = '555-7777';
```

1. The hashing algorithm generates the row hash for this primary index access and finds that rows having the NUPI value 555-7777 hash to AMP 4.
2. The Dispatcher sends an AMP retrieval step directly to AMP 4, where the file system retrieves two matching rows.

The first matching row has row hash=778 and UV=3, while the second matching row has row hash=778 and UV=7. The row hash values are identical because this primary index is nonunique.

3. The file system reads the requested rows and returns them to the requestor.

Cust=95, Name=Peters, Phone=555-7777

Cust=56, Name=Smith, Phone=555-7777

Query 2

The second query uses the USI column Cust as the WHERE clause attribute, with the requested value being 95.

```
SELECT *
FROM Customer
WHERE Cust = 95;
```

1. The hashing algorithm generates the row hash for this USI access.
The hash map indicates that an index row having the value 95 hashes to the customer USI subtable on AMP 3 with a row hash=588.
2. The Dispatcher sends an AMP retrieval step directly to AMP 3, where the file system reads the subtable row having the row hash=588 to determine which AMP owns the base table row.
3. The file system retrieves the base table row hash 778 and uniqueness value 3 from the USI subtable row and determines that the requested base table row is stored on AMP 4.
4. The retrieval directive is passed to AMP 4 where the row for customer number 95, having rowID value 778,3 is located.
5. The file system reads the requested row and returns it to the requestor.

Cust=95, Name=Peters, Phone=555-7777

Query 3

The third query uses the NUSI column, name, as the WHERE clause attribute, with the requested value being the name column value Smith.

```
SELECT *
FROM Customer
WHERE Name = 'Smith';
```

1. The Dispatcher broadcasts an AMP retrieval step containing the NUSI value Smith to all AMPs.
2. The file system scans the NUSI subtable with row hash 432 and selects index rows for Smith in customer on each AMP, retrieving all the base table rowIDs and uniqueness values associated with the name column value of Smith and row hash value 432.

The steps are processed in parallel and one row is located on each AMP.

- AMP1 (NUSI row hash=432, UV=8; Base table row hash=640, UV=1)
- AMP2 (NUSI row hash=432, UV=3; Base table row hash=884, UV=1)
- AMP3 (NUSI row hash=432, UV=1; Base table row hash=147, UV=1)
- AMP4 (NUSI row hash=432, UV=5; Base table row hash=778, UV=7)

3. The file system directly retrieves the base table rows on each AMP in parallel and returns them to the requestor.

Cust=40, Name=Smith, Phone=222-3333

Cust=74, Name=Smith, Phone=555-6666

Cust=49, Name=Smith, Phone=111-6666

Cust=56, Name=Smith, Phone=555-7777

Join and Hash Indexes

This section describes join and hash indexes. These indexes permit you to undertake a wide variety of physical denormalizations of the database without affecting the normalization of the logical and physical models.

You should always define an equivalent single-table join index rather than a hash index so as not to depend on the default choices that are made for a hash index to be the correct choices. Also, hash indexes have some limitations such as multivalued compression is not carried over to a hash index from the base table.

Other topics include information on how to estimate the overhead of various join indexes, Optimizer criteria for selecting a join index, special storage options offered by join indexes, and numerous examples.

See [Design Issues for Tactical Queries](#) for a description of the special design considerations that must be evaluated for using hash and join indexes to support tactical queries.

For information about design issues related to join indexes defined on temporal tables and system-defined join indexes, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Join Indexes

Join indexes are designed to permit queries (join queries in the case of multitable join indexes) to be resolved by accessing the index instead of accessing, and possibly joining, their underlying base tables.

Multitable join indexes are useful for queries where the index contains all the columns referenced by one or more joins, thereby allowing the index to cover that part of the query, making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index (some vendors refer to this as index-only access). Note that a join index that is defined with an expression in its select list provides less coverage than a join index that is defined using a base column (see [Restrictions on Partial Covering by Join Indexes](#)).

Even if a join index does not completely cover a query, the Optimizer can use it to join to its underlying base tables in a way that provides better query optimization than scanning the base tables for all the columns specified in the request (see [Partial Query Coverage](#)).

From the point of view of a database designer, the multitable join index permits great adaptability because it provides the flexibility of normalization while at the same time offering the opportunity to create alternative, denormalized virtual data models, providing what might be called materialized views of the database.

Depending on how the index is defined, single-table join indexes can also be useful for queries where the index contains only some of the columns referenced in the statement. This situation is referred to as a partial covering of the query. Multitable join indexes can also be used to partially cover a query for one or more of the tables defined in the join index.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. These indexes play the role of prejoin and summary tables without denormalizing the logical design of the database

and without incurring the update anomalies and performance problems presented by denormalized tables. While it is true that denormalization often enhances the performance of a particular query or family of queries, it can and often does make other queries perform more poorly.

Unlike traditional indexes, join indexes are not required to store pointers to their associated base table rows. Instead, they are generally used as a fast path final access point that eliminates the need to access and join the base tables they represent. They substitute for, rather than point to, base table rows. The only exception to this is the case where an index partially covers a query. If the index is defined using either the ROWID keyword, the UPI of its base table, or a USI on the base table as one of its columns, then it can be used to join with the base table to cover the query. A join index defined in this way is sometimes called a global index or global join index. Note that you can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

In this case, you might create a join index to contain only the join column and the ROWID or the UPI column set of the table. The process is as follows.

1. The join index is joined with the other table and any selection conditions that can be evaluated during the join to eliminate disqualified rows are applied.
2. The result of the join is stored in a temporary table and redistributed based on the ROWID, UPI, or USI of the base table to perform a join with the base table.

This join can happen at different points in the query plan, depending on estimated costs. For example, the Optimizer might determine that it is cheaper to perform a join with another table involved in the query before joining to the base table.

Depending on the workloads they are designed to support, you can create join indexes that have either partitioned or nonpartitioned primary indexes or are column-partitioned. You can also create column-partitioned join indexes that have no primary index. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for the usage rules for column-partitioned join indexes.

Rules for Using the ROWID Keyword in a Join Index Definition

- The ROWID keyword can only be used in a join index definition.
- You can optionally specify ROWID for a base table in the select list of a join index definition. For a column-partitioned join index, the ROWID for the base table in the select list of the join index definition is required.

If you reference multiple tables in the join index definition, then you must fully qualify each ROWID specification.

- You can reference an alias name for ROWID, or the keyword ROWID itself if no correlation name has been specified for it, in the primary index definition or in a secondary index defined for the join index in its index clause.

This does not mean that you can reference a ROWID or its alias in a secondary index defined separately with a CREATE INDEX statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144) after the join index has been created.

- An alias is required for ROWID if the join index is column partitioned.

- If you reference a ROWID alias in the select list of a join index definition, then you can also reference that alias in a CREATE INDEX statement that creates a secondary index on the join index.
- Aliases are required to resolve any column name or ROWID ambiguities in the select list of a join index definition.

An example is the situation where you specify ROWID for more than one base table in the index definition.

Also see the description of the CREATE JOIN INDEX statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Rules for Using the System-Derived PARTITION Column in a Hash or Join Index Definition

You cannot specify the system-derived PARTITION column in any hash or join index definition.

You can specify a user-named column named "partition" in an index definition.

Rules for Using Join Indexes on Load-Isolated Tables

When you create a join index, if any of the base tables are load isolated, then the JI is marked as a table with load isolation. This means that the load-isolated join index is 8 bytes wider because each row contains a RowLoadID. Modification operations on a join index are driven by modifications to the base table and the JI definition.

A load operation on the base table implicitly starts a load operation on the JI. A load commit on the load-isolated base table commits the load on the underlying JI. A compressed join index is not supported on a load-isolated table.

Default Column Multivalue Compression for Join Index Columns When the Referenced Base Table Column Is Compressed

When you create a join index, Vantage automatically transfers any column multivalue compression defined on the base table, with a few exceptions, to the join index definition. In contrast, hash indexes do not inherit the multivalue compression defined for the columns of their parent base table.

The following rules and restrictions apply to automatically transferring the column compression characteristics of a base table to its underlying join index columns. All of these rules and restrictions must be met for base table column multivalue compression to transfer to the join index definition:

- Base table column multivalue compression transfers to a join index definition even if there is an alias name specified for the columns in the join index definition.
- Base table column compression transfers to a multivalue join index definition only up to the point that the maximum table header length of the join index is exceeded.

The CREATE JOIN INDEX request does not abort at that point, but the transfer of column multivalue compression from the base table to the join index definition stops.

- Base table column multivalue compression does not transfer to a join index definition if the column is a component of the primary index for the join index.

- Base table column multivalue compression does not transfer to a join index definition for any of the columns that are components of a partitioned primary index, a partitioning expression for a PPI join index, or a partitioning expression for a column-partitioned join index.
- Base table column multivalue compression does not transfer to a join index column if the column is specified as the argument for any of the following functions.
 - COUNT
 - EXTRACT
 - MIN
 - MAX
 - SUM
- Base table column multivalue compression does not transfer to a column in a compressed join index that has indexes defined on its column_1 and column_2 sets.
- Base table column multivalue compression does not transfer to a join index definition if the column is a component of an ORDER BY clause in the join index definition.

Compression of Join Indexes at the Block Level

Because join indexes are primary tables, they can be compressed at the block level. See [Compressing Data Loaded into Empty Subtables Set to AUTOTEMP](#), and [Compression Types Supported by Vantage](#) for more information about data block compression.

Summary of Join Index Functions

A join index always has at least one of the following functions.

- Replicates all, or a vertical subset, of a single base table and partitions its rows using a different primary index (or a different column partitioning if the base table is a NoPI column-partitioned table) than the base table, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.

Note:

A partitioning expression for a row-partitioned join index cannot contain a row-level security constraint column.

- Joins multiple tables (optionally with aggregation) in a prejoin table.
- Aggregates one or more columns of a single table as a summary table.

Join indexes are updated automatically, so the only administrative task a DBA must perform is to keep the statistics on multitable join index columns and their indexes current. For non-sparse single-table join indexes, the best policy is to use base table statistics rather than to collect statistics directly on the columns of the index.

The recommended practice for recollecting statistics is to set appropriate thresholds for recollection using the THRESHOLD options of the COLLECT STATISTICS statement. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You cannot collect statistics on complex expressions specified in a base table definition. However, if you frequently submit queries that specify complex base table predicate expressions, you can create a single-table join index or hash index that specifies those frequently used predicate expressions in its select list or column list, respectively, and then collect statistics on the expression defined as a simple column in your index.

There are several specific cases where join index statistics can provide more accurate cardinality estimates than are otherwise available for base table predicates written using complex date expressions.

- The case where an EXTRACT expression specified in a query predicate can be matched with a join index predicate.
- The case where an EXTRACT/DATE expression specified in a query predicate condition can be mapped to an expression specified in the select list of a join index.

The Optimizer uses expression mapping when it detects an identical query expression or a matching query expression subset within a non-matching predicate. When this occurs, the Optimizer maps the predicate to the identical column of the join index, which enables it to use the statistics collected on the join index column to estimate the cardinality of the expression result.

When you create a single-table join index that specifies a complex expression, Vantage transforms the expression into a simple join index column. This enables the Optimizer to map the statistics collected on those complex expressions to the base table to facilitate single-table cardinality estimates or to match the predicates (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details).

Note that the derived statistics framework supports bidirectional inheritance of statistics between a non-sparse single-table join index and the base table it supports (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details), so the entity on which statistics are collected is no longer as important as it once was.

Similarities of Join Indexes to User Data Tables

In many respects, hash and join indexes are identical to base user data tables.

For example, you can do the following things with a join index.

- Create a unique or nonunique primary index, either a PPI or a nonpartitioned primary index, on its columns.

UPIs are supported only for uncompressed single-table join indexes without an ORDER BY clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Create a column-partitioned, single-table uncompressed, non-aggregate join index. Such a join index can be sparse join index.
- Create nonunique secondary indexes on its columns.
- Perform any of the following statements against it.
 - COLLECT STATISTICS (Optimizer Form)
 - DROP JOIN INDEX
 - DROP STATISTICS (Optimizer Form)

- `HELP JOIN INDEX`
- `SHOW JOIN INDEX`
- Specify UDT and BEGIN and END bound functions on Period or derived Period columns in its definition.
- Specify row-level security constraint columns in its definition.

You can do this only if both of the following criteria are true:

- The index references a maximum of one row-level security-protected base table.
- All of the row-level security constraint columns defined in the base table are included in the join index definition.

Note:

You cannot specify row-level security constraint columns in a partitioning expression for a row-partitioned join index.

- Specify any valid expressions in the select list and WHERE clause when the expressions reference at least one column.

The following expression types are not valid in a join index definition:

- OLAP expressions
- UDF expressions
- Built-in functions that are explicitly not valid such as `DEFAULT` and `PARTITION`.

Note that a join index defined with an expression in its select list provides less coverage than a join index that is defined using a base column (see [Restrictions on Partial Covering by Join Indexes](#)).

Unlike base tables, you cannot do the following things with join indexes:

- Create a join index on a join index.
- Query or update join index rows.

For example, if `ordCustIdx` is a join index, then the following query is not legal:

```
SELECT o_status, o_date, o_comment
FROM ordCustIdx;
```

- Create a USI on its columns.
- Define multivalued or algorithmic compression on its columns.

If multivalued or algorithmic compression are defined on any columns of its parent base table set, however, a join index does inherit that compression under most circumstances (see [Default Column Multivalued Compression for Join Index Columns When the Referenced Base Table Column Is Compressed](#)).

Join Index Applications

Join indexes are useful for queries where the index table contains all the columns referenced by one or more joins, thereby allowing the Optimizer to cover all or part of the query by planning to access the index

rather than its underlying base tables. An index that supplies all the columns requested by a query is said to cover that query and, for obvious reasons, is often referred to as a covering index. Some vendors refer to this as *index-only access*. A join index can be particularly useful for queries that access both nonpartitioned and column-partitioned tables (see [NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes](#)). If either a nonpartitioned NoPI table or a column-partitioned NoPI table has no secondary indexes, a covering join index is the only way to access its rows without using a full-table scan. Be aware that join indexes can slow the loading of rows into a table using Teradata Parallel Data Load array INSERT operations.

The Optimizer can also use join indexes that only cover a query partially if the index is defined properly (see [Partial Query Coverage](#)). Note that query covering is not restricted to join indexes: other indexes can also cover queries either in whole or in part.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. For these applications, join indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and ad hoc query performance issues frequently presented by denormalized tables.

You can create join indexes that limit the number of rows in the index to only those that are accessed when a frequently run query references a small, well-known subset of the rows of a large base table. You create this type of join index by specifying a constant expression as the RHS of the WHERE clause, which narrowly filters the rows included in the join index. This is known as a sparse join index.

You can also create join indexes that have a partitioned primary index (you can only define a PPI for a join index if the index is not row-compressed) or that are column-partitioned join indexes. Note that you cannot create a nonpartitioned NoPI join index. PPI join indexes are useful for covering range queries (see [Designing for Range Queries: Guidelines for Choosing Between a PPI and a Value-Ordered NUSI](#)), providing excellent performance by means of row partition elimination (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142).

See [Sparse Join Indexes and Tactical Queries](#) for specific design issues related to join index support for tactical queries.

Partial Query Coverage

Partial query coverage allows join indexes whose columns do not match an entire query to be used to cover a subset of it. For example, one or two tables specified by the query might be covered by the join index, but the entire request is not. In some situations, there might be a number of commonly performed queries that join several tables where each of the queries joins two tables, say t1 and t2, on the same columns. For this situation, you can create a join index to join t1 and t2, and the Optimizer can use that join index for any queries that need to perform that join.

Partial query coverage also allows join indexes that contain only a subset of the columns of a base table referenced in the query to cover the query if that join index can be joined to the base table to retrieve additional referenced columns (this form of partial coverage is also used to implement hash indexes: see [Hash Indexes](#)).

For example, suppose there is a large table that needs to be joined frequently with another table on a column that is not the distributing column of the table. You can define a join index that redistributes the base table

by the join column. However, because of the large number of rows and columns that need to be projected into the join index, the extra disk storage required does not allow the creation of such a join index.

You can also define a join index in such a way that its partial coverage of a query can be extended further by joining with a parent base table to pick up any columns requested by the query but not referenced in the join index definition.

Such a join index, sometimes called a global index or global join index, is defined with one of the following elements, which the Optimizer can use to join it with a parent base table to extend its coverage:

- One of the columns in the index definition is the keyword ROWID. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.
- The column set defining the UPI of the underlying base table.
- The underlying base table.

See [Restrictions on Partial Covering by Join Indexes](#) for an example of a global join index.

Designing for Range Queries: Guidelines for Choosing Between a PPI and a Value-Ordered NUSI

Row-partitioned primary indexes and value-ordered NUSIs are both designed to optimize the performance of range queries. Because you cannot define both a PPI and a value-ordered primary index on the same join index (nor can you define a PPI for a row-compressed join index), you must determine which is more likely to enhance the performance of a given query workload for those situations that exclude using a join index with a value-ordered primary index.

In general, a value-ordered primary index is the preferred choice if it meets the restriction of a 4-byte maximum column size. Note that this cannot be used for an MLPPI because MLPPIs do not support value-ordered NUSIs by definition.

You might want to consider creating a multilevel partitioning on the base table as an alternative if the usage would be roughly equivalent to a value-ordered NUSI on a join index plus a non-value-ordered NUSI.

In nearly all cases, a join index with a value-ordered primary index is the preferred choice over a value-ordered NUSI.

You should consider the following guidelines when determining whether to design a join index for a particular workload using row partitioning on the join index or using an nonpartitioned primary index and adding a value-ordered NUSI to create a value-ordered access path to the rows:

- It is better to use row partitioning on the join index than to use a value-ordered NUSI and an nonpartitioned primary index on the join index.
- If row compression is defined on the join index, then you cannot define partitioning for it.

For this scenario, a value-ordered primary index or value-ordered NUSI are your only join index design choice possibilities for optimizing range queries.

- Each time an underlying base table for a join index is updated, the join index also must be updated.

If there is a value-ordered NUSI defined on a join index, then it, too, must be updated each time the base table (and join index) rows are updated.

You can avoid this additional maintenance when you define the join index with row partitioning. Row partition elimination makes updating a join index row even faster than the equivalent update operation against a nonpartitioned primary index join index. "Update" is used in a generic sense to include the delete and update operations performed by the DELETE, MERGE, and UPDATE SQL statements, but excluding insert operations performed by the SQL INSERT and MERGE statements, where row partition elimination is not a factor.

Row partitioning can also improve insert operations if the inserted rows are clustered in the data blocks corresponding to the partitions. In this case, the number of data blocks read and written is reduced compared with the nonpartitioned primary index join index case where the inserted rows are scattered among all the data blocks. Because of the way that the rows of a column-partitioned join index are distributed to the AMPs, you should not expect to see the positive effects of data block clustering for singleton INSERT requests. However, you should see very positive effects for large INSERT ... SELECT loads into column-partitioned join indexes when the base table is loaded with rows using an INSERT ... SELECT request.

- Row-partitioned join indexes offer the benefit of providing direct access to join index rows, while a value-ordered NUSI does not.

Using a NUSI to access rows is always an indirect operation, touching the NUSI subtable before being able to go back to the join index to access a row set.

Besides offering a direct path for row access, row partitioning provides a means for attaining better join and aggregation strategies on the primary index of the join index.

- If you specify the primary index column set in the query, row partitioning and join indexes offer the additional benefit of enhanced direct join index row access using row partition elimination.

The comparative row access times ultimately depend on the selectivity of a particular value-ordered NUSI, the join index row size, and whether a value-ordered NUSI covers a given query or not.

- If a join index partitioning column has more than 65,535 unique values, and the query workload you are designing the index for probes for a specific value, a value-ordered NUSI is likely to be more selective than a join index partitioned on that column.

Note that because NUSI access is indirect, the system might read entire data blocks to retrieve one or a few rows using a NUSI, while row-partitioned index access is direct, with like rows being clustered together, so row-partitioned join index read operations are usually far more efficient than value-ordered NUSI read operations.

Collecting Statistics on a Join Index

Issues concerning collecting statistics on the indexes of a join index are documented in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184. Also see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about using single-table join index statistics to make cardinality estimates that cannot otherwise be made with the same level of confidence.

Fallback With Join Indexes

You can define fallback for join indexes. The criteria for deciding whether to define a join index with fallback are similar to those used for deciding whether to define fallback on base tables.

If you do not define fallback for a join index and an AMP is down, then the following additional criteria become critical:

- The join index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the join index is defined cannot be updated.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

Limits for Hash and Join Indexes

- Tables can have up to 32 secondary, hash, and join indexes.

These 32 indexes can be any combination of secondary, hash, and join indexes, including the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints.

Each multicolumn NUSI that specifies an ORDER BY clause counts as two consecutive indexes in this calculation.

- Index columns cannot have XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, JSON, ARRAY, or VARRAY. Hash indexed columns cannot have Period data types.
- Join index columns can have Period data types and can include expressions composed of system and user-defined functions and methods, including the use of the BEGIN, END, and P_INTERSECT built-in functions on a Period data type column to compose an expression in the projection list and a single-table condition in its WHERE or ON clauses.
- Both hash and join indexes can be created on a row-level security-protected table only if all of the following criteria are met.
 - The hash or join index references a maximum of one row-level security-protected table.
 - All of the row-level security constraint columns in the indexed table are included in the hash or join index definition.

Further Information

Consult the following documents for more detailed information on creating and using join indexes to enhance the performance of your database applications:

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Temporal Table Support*, B035-1182
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

Using Join Indexes

You can create a join index to perform any of the following operations:

- Join multiple tables, optionally with aggregation, in a prejoin table.
- Replicate all or a vertical subset of a single base table and distribute its rows by a primary index on a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Aggregate one or more columns of a single table or the join results of multiple tables in a summary table.
- Support querying only those rows that satisfy the conditions specified by its WHERE clause. This is known as a sparse join index.
- If the index has a unique primary index, and a request specifies an equality condition on the columns that define the primary index for the index, then the index can be used for the access path in two-AMP join plans similarly to how USIs are used.

The guidelines for creating a join index are the same as those for defining any regular join query that is frequently executed or whose performance is critical. The only difference is that for a join index the join result is stored as a subtable and automatically maintained by Vantage.

Performance and Join Indexes

Requests that can use join indexes can run many times faster than queries that do not use them. Performance improves whenever the Optimizer can rewrite a request to use a join index instead of the base tables specified by the query.

A join index is most useful when its columns can cover most or all of the requirements in a request. For example, the Optimizer might consider using a covering index instead of performing a merge join.

Covering indexes improve the speed of join queries. The extent of improvement can be dramatic, especially for requests involving complex, large-table, and multiple-table joins. The extent of the improvement depends on how often an index can be used to rewrite a query.

In-place join indexes, where the columns of the covering index and the columns of the table to which it is to be joined both reside on the same AMP, outperform indexes that require row redistribution. An in-place, covering, aggregate join index that replaces 2 or more large tables in requests with complex joins, aggregations, and redistributions can enable a request to run hundreds of times faster than it would otherwise.

Partial Covering Multitable Join Indexes

Vantage optimizes queries to use a join index on a set of joined tables even if the index does not completely cover the columns referenced in the table if the following things are true.

- The index includes either the Row ID or the columns of a unique index on the table containing a non-covered column referenced by the query.
- The cost of such a plan is less than other competing query plans.

A partial covering multitable join index provides some of the query improvement benefits that covering join indexes offer without replicating all of the table columns required to cover requests in the join index, but

an additional overhead from having to access base table rows to retrieve column values occurs when a non-covered column is specified in a request.

Covering Bind Terms

A bind term is a condition that connects an outer query and a subquery. If the connecting condition of a subquery is IN and the column it is connecting to in the subquery is unique, you can define a join index on the bind term columns. This provides one more type of index for the Optimizer to consider using in place of multiple base tables.

Using Single-Table Join Indexes

Single-table join indexes are useful in tactical applications because they can support alternative access paths to data. This is a good approach to consider when a tactical query carries a value in an equality condition for a column, such as a customer phone number, that is in the table but is not its primary index. This might be a customer key, for example. A single-table join index can be constructed using the available non-indexed column, the customer phone number, as its primary index, thereby enabling single-AMP access to the data and avoiding more costly all-AMP non-primary index access to the base table.

Single-table join indexes are also valuable when your applications often join the same large tables, but their join columns are such that some row redistribution is required. A single-table join index can be defined to contain the data required from one of the tables, but using a primary index based on the FK of the table, preferably the primary index of the table to which it is to be joined. A single-table join index can also be used as a virtual vertical partitioning of a base table, creating an index subtable that contains frequently accessed columns from a table with many columns that generally are not accessed.

Use of such an index greatly facilitates join processing of large tables, because the single-table index and the table with the matching primary index both hash to the same AMP.

The Optimizer evaluates whether a single-table join index can replace or partially cover its base table even when the base table is referenced in a subquery unless the index is compressed and the join is complex, such as an outer join or correlated subquery join.

Using Outer Joins to Define Join Indexes

If there is a need for a non-aggregate multitable join index between several large tables, considering using an outer-join to define your join index. This approach offers the following benefits.

- The Optimizer will consider a join index defined using an outer-join for queries that reference only the outer tables of the defining outer join.
- The join index preserves the unmatched rows in the outer join within the join index structure.

Defining Join Indexes with Inequality Conditions

You can define join indexes using inequality conditions.

To define inequality conditions between 2 columns of the same type, either from the same table or from two different tables, you must AND them with the other join conditions.

This enables the Optimizer to resolve a request using a join index more frequently than would otherwise be possible, avoiding the need to access base data tables to retrieve the required data.

Defining Join Indexes on UDT Columns and Expressions

You can define join indexes on the following UDT columns and expressions in the select list and in single-table conditions in the WHERE and ON clauses.

- UDT columns, including using a method that is associated with a UDT column
- The following types of expressions.
 - Non-aggregate expressions
 - Non-OLAP expressions
 - Non-UDF expressions

This general support for UDT columns and expressions enables you to specify Period data type columns and BEGIN, END and P_INTERSECT expressions on a Period data type in the select list, WHERE clause, and ON clause of a join index definition.

Refreshing Join Indexes

The ALTER TABLE TO CURRENT statement enables you to refresh the content of a join index without having to drop it and recreate it.

The efficiency of the ALTER TABLE TO CURRENT alternative compared with dropping and recreating a join index depends on how often an ALTER TABLE TO CURRENT request is executed and the type of current date condition defined in the join index.

If the join index is refreshed infrequently and the current date condition requires a large volume of old rows to be removed and a large volume of new rows to be inserted, it might be more efficient to drop and recreate the join index.

Using Aggregate Join Indexes

Aggregate join indexes offer an extremely efficient, cost-effective method of resolving requests that frequently specify the same aggregation operations on the same column or columns. When aggregate join indexes are available, the system does not have to repeat aggregation calculations for every request.

You can define an aggregate join index on two or more tables or on a single table. A single-table aggregate join index includes:

- A subset of the columns in a base table
- Additional columns for the aggregate summaries of the base-table columns

You can create an aggregate join index using the GROUP BY clause and the following built-in aggregate functions.

- SUM
- COUNT
- MAX
- MIN

The following restrictions apply to defining an aggregate join index.

- Only the COUNT, MAX, MIN, and SUM aggregate functions are valid in any combination.
COUNT DISTINCT and SUM DISTINCT are not valid.
- To avoid overflow, always type the COUNT, MAX, MIN, and SUM columns in an aggregate join index definition as FLOAT.

Vantage enforces this restriction as follows.

IF you ...	THEN Vantage ...
do not define an explicit data type for a COUNT, MAX, MIN, or SUM column	assigns the FLOAT data type to it automatically.
define a COUNT, MAX, MIN, or SUM column as anything other than FLOAT	returns an error and does not create the aggregate join index.

Many aggregate functions are based on the SUM, MAX, MIN, and COUNT functions, so even though you cannot specify many individual aggregate functions in an aggregate join index, you can combine these 4 functions in a number of ways to create an aggregate join index to resolve requests that use more complicated aggregate functions.

A simple example is using the COUNT and SUM functions to compute an average.

$$AVG(x) = \frac{SUM(x)}{COUNT(x)}$$

Join Indexes and the Optimizer

For each base table in a query, the Optimizer performs certain processing phases to decide how a database operation that uses a join index is to be processed.

In this phase...	The optimizer...
Qualification	evaluates up to 10 join indexes to choose the one with the lowest cost. Qualification for the best plan includes one or more of the following benefits: <ul style="list-style-type: none"> • Smallest size to process • Most appropriate distribution • Ability to take advantage of covered fields within the join index
Analysis of results	determines if this plan will result in unique results, analyzing only those tables in the query that are used in the join index.

Subsequent action depends on analysis of the results.

IF the results are...	THEN the Optimizer...
unique	skips the sort-delete steps used to remove duplicates.
nonunique	determines whether eliminating all duplicates can still produce a valid plan, recognizing any case where the following things are true. <ul style="list-style-type: none"> • No <i>column_name</i> parenthetical clause exists • All logical rows will be accessed

System Processing of Join Indexes

The Optimizer does the following when it rewrites requests using a join index.

- Selects cost-based query rewrites using the best available aggregate join index when several possible aggregate join indexes are available.
- Provides a larger number of opportunities to perform cost-based rewrites of requests using aggregate join indexes for queries with subqueries, spooled derived tables, outer joins, COUNT(DISTINCT), and extended grouping sets.

When you create multiple aggregate join indexes, the creation of the current aggregate join index makes use of an existing aggregate join index that is most efficient for the calculation of the aggregate join index being created so that the CREATE JOIN INDEX request has better performance.

With the existence of multiple join indexes, including aggregate join indexes and non-aggregate join indexes, aggregate queries perform better with the cost-based rewrite and more chances to use an aggregate join index.

- Uses join indexes with Partial GROUP BY optimizations during join planning, making it possible to produce better join plans.

Join Index Optimizations

The Optimizer uses join indexes in several ways, including the following.

- Selects cost-based query rewrites using the best available aggregate join index when several possible aggregate join indexes are available.
- Provides a larger number of opportunities to perform cost-based rewrites of requests using aggregate join indexes for queries with subqueries, spooled derived tables, outer joins, COUNT(DISTINCT) operations, and extended grouping sets.

When a user creates multiple aggregate join indexes, the creation of the current aggregate join index makes use of an existing aggregate join index that is the most efficient for the calculation of this aggregate join index so that the CREATE JOIN INDEX request will have better performance.

With the existence of multiple join indexes, including aggregate join indexes and non-aggregate join indexes, aggregate requests perform better with the cost-based rewrite and more chances to use an aggregate join index.

- Uses join indexes with Partial GROUP BY optimizations during join planning, making it possible to produce better join plans.

Protecting a Join Index with Fallback

You can define fallback protection for a simple or aggregate join index.

With fallback, you can access a join index and the base table it references if an AMP fails, with little impact on performance.

Without fallback, an AMP failure has significant impact on both availability and performance as follows.

- You cannot update the base table referenced by a join index even if that base table is defined with fallback.
- The Optimizer cannot access a join index on a down AMP to create query plans. Performance can be degraded significantly when this occurs.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

The cost of having fallback for a join index when executing a DML request that modifies a base table referenced by the join index is a slight increase in processing to maintain the fallback copy of the join index.

Collecting Statistics for Join Indexes

Hash indexes and single-table join indexes that are not defined as sparse join indexes inherit all statistics from their base table, including dynamic AMP samples and collected statistics.

Only sparse join indexes and multitable join indexes require statistics collection. It is particularly important that statistics be collected on the sparse-defining column in the WHERE clause of a sparse join index or the Optimizer might not select the sparse join index for use.

Consider collecting statistics to improve performance during the following operations.

- Creation of a join index
- Update maintenance of a join index

You need to submit separate COLLECT STATISTICS requests for the columns in the join index and the source columns in the base tables. This does not have a very high cost because Vantage can collect statistics while queries are accessing the underlying base tables of a join index.

Costing Considerations for Join Indexes

Join indexes, like secondary indexes, incur both space and maintenance costs. For example, INSERT, UPDATE, and DELETE operations must be performed twice: once for the base table and once for the join index.

Space Costs for Join Indexes

The following formula estimates the space overhead required for a join index.

Join Index Size = $U \times (F + O + (R \times A))$

where:

Parameter	Description
F	Length of the fixed column <i>join_index_column_1</i>
R	Length of a single repeating column <i>join_index_column_2</i>
A	Average number of repeated fields for a given value in <i>join_index_column_1</i>
U	Number of unique values in the specified <i>join_index_column_1</i>
O	Row overhead (assume 14 bytes)

Updates to the base tables can cause a physical join index row to split into multiple rows. The newly formed rows each have the same fixed field value but contain a different list of repeated field values. This applies specifically when the compressed join index format is being used.

The system, however, does not automatically recombine logically related split rows. To re-compact such rows, you must drop and recreate the join index.

Maintenance Costs for Join Indexes

The use of a join index entails the following.

- Initial time consumed to calculate and create the index
- Whenever a value in a join index column of the base table is updated, the join index must also be updated, including any required aggregation or pre-join effort.

However, if join indexes are suited to your applications, the improvements in request performance can far outweigh the costs.

Join indexes are maintained by generating additional AMP steps in the base table update execution plan. Those join indexes defined with outer joins usually require additional steps to maintain any unmatched rows.

Expect a single-table join index INSERT operation to have similar maintenance overhead as would an insert operation with an equivalent NUSI. UPDATE or DELETE operations, however, might incur greater overhead with a single-table join index, unless a value for the primary index of the join index is available at the time of the update.

Overhead for an in-place aggregate join index can be perhaps 3 times more expensive than maintaining the same table without that index. For an aggregate join index that redistributes rows, the maintenance overhead can be several times as expensive.

Maintenance overhead for multitable join indexes without aggregates can be small or very large, depending on the pre-join effort involved in constructing or changing a join index row. This could be up to 20 times or more expensive than maintaining the table without the index. The overhead is greater at higher hits per block, where *hits* means the number of rows in a block are touched.

Since Vantage writes a block only once regardless of the number of rows modified, as the number of hits per block increases:

- The CPU path per transaction decreases (faster for the case with no join index than for the case with a join index)
- Maintenance overhead for aggregate join indexes decreases significantly

If a DELETE or UPDATE request specifies a search condition on the primary index or secondary index of a join index, the join index may be directly searched for the qualifying rows and modified accordingly.

This direct-update approach is employed when the request adheres to these requirements:

- A primary index or secondary index access path to the join index
- If a join_index_column_2 is defined, little or no modification to the join_index_column_1 columns
- No modifications to the join condition columns in the join index definition
- No modifications to the primary index columns of the join index

It is not necessary to drop the join index before a backup. It is important, however, to drop join indexes before the underlying tables and databases are restored, should a restore ever be required. Otherwise an error is reported and the restore will not be done.

Join Indexes Versus NUSIs

A join index offers the same benefits as a standard secondary index in that it, like the standard secondary index, has the following properties.

- Optional
- User defined
- Maintained by the system
- Transparent to end users
- Immediately available to the Optimizer
- If a covering index, considered by the Optimizer for a merge join

However, a join index offers the following performance benefits over a NUSI.

IF a join index is...	THEN performance improves by...
defined using joins on one or more columns from two or more base tables	eliminating the need to perform the join step every time a joining query is processed.
used for direct access in place of some or all of its base tables, if the Optimizer determines that it covers most or all of the query.	eliminating the I/Os and resource usage required to access the base tables.
limited to only certain data types of your choice, such as Date	allowing direct access to the join index rows within the specified value-order range.
a single-table join index with a FK primary index	reducing I/Os and message traffic because row redistribution is not required, since the following are hashed to the same AMP: <ul style="list-style-type: none"> • A single-table join index having a primary index based on the base table foreign key. • The table with the column set making up the foreign key.

IF a join index is...	THEN performance improves by...
defined with an outer join	<ul style="list-style-type: none"> • Giving the same performance benefits as a single-table join index, for queries that reference only outer tables. • Preserving unmatched rows.
created using aggregates	eliminating both the aggregate calculations and the join step for every query requiring the join and aggregate.

For more information on the syntax, applications, restrictions, and benefits of join indexes, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Join Index Design Tips

This topic describes various tips that you can use to optimize the functionality of your join indexes.

When You Should Consider Defining a Join Index

Join indexes are not suited for all applications and situations. The usefulness of a join index, like that of any other index, depends on the type of work it is designed to perform. Always prototype any join index and evaluate its usefulness to the applications it is designed to support before adding it to your production environment. The overhead of updating join index tables can outweigh their benefit in some situations.

The following situations all make a join index a likely performance enhancer:

- Frequent joins of large tables with other large or moderately-sized tables that result in a significant number of the rows from both tables being joined.
- Frequent joins of tables of high degree (having many columns) for which the same relatively small set of columns is repeatedly requested.
- An alternate partitioning sequence for a vertical subset of data in one of the base tables (a so-called *single-table join index*) would remove the necessity of redistributing rows for a frequently made join.
- The overhead in time and storage capacity for the creation and maintenance of a join index does not outweigh its retrieval benefits.
- The performance of frequent range queries requiring joins of large tables with other large or moderately-sized tables that result in a significant number of the rows from both tables being joined.
- A row-partitioned join index can enhance the performance of queries if you specify an equality or range constraint on the partitioning column set. For example, a single-table row-partitioned join index can take advantage of row partition elimination to improve both the performance of a query retrieving rows from itself.

Be aware that you cannot define row partitioning for a row-compressed join index.

- If a frequently run query specifies a complex expression in its predicate, consider creating a single-table join index or a hash index on the table that includes that expression in the select list or column list, respectively, of its definition. Although you cannot collect statistics on complex base table expressions, creating a single-table join using the expression transforms it into a simple column, and you can then collect statistics on that column. The Optimizer can then use those statistics to estimate the single-table

cardinality of evaluations of the expression in a query predicate that specifies the expression using a base table column. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

- Most queries against a column-partitioned table or join index are expected to be very selective on a variable subset of columns, and project a variable subset of the columns where the subset of accessed columns is less than 10% of the column partitions for any particular query.

Sometimes you just need to experiment.

For example, application of a row-partitioned join index might be for queries that involve row-partitioned base tables. However, if the base table is not a row-partitioned table, but is designed to handle efficient joins on the primary index, it is also conceivable that a row-partitioned join index might be defined to provide an alternative organization of the data for optimal access based on row partitions. This is only valid if the join index is not row-compressed. Partitioning is not valid for row-compressed join indexes.

See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for further information about PPI join indexes.

Using Outer Joins to Define Simple Join Indexes

Because join indexes generated from inner joins do not preserve unmatched rows, you should consider using outer joins to define simple join indexes, noting the following restrictions.

- Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.
- Outer joins are not supported under any circumstances for aggregate join indexes.

Defining join indexes on outer joins enables them to satisfy queries with fewer join conditions than those used to generate the index. See [Defining a Simple Join Index on a Binary Join Result](#) and [Using Outer Joins to Define Join Indexes](#) for a demonstration of this property.

Also see [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#) and related examples.

Collecting Statistics on the Columns and Indexes for Join Indexes

Keeping fresh statistics on join indexes is just as critical for join indexes as it is for base tables for optimizing the query plans generated by the Optimizer.

This is particularly true if you have created a single-table join index whose definition includes a complex expression in its select list and whose statistics are to be used to make more accurate single-table cardinality estimates when a query that specifies a matching expression in its predicate is run against the table on which the join index is defined. The Optimizer can also use expression mapping when it detects an expression in a query predicate within a non-matching predicate. In this case, the Optimizer maps to the join index column that is defined using the matched expression. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

You cannot collect statistics on the PARTITION column of a PPI join index.

Basing a Join Index on Foreign Key-Primary Key Equality Conditions

To avoid storing redundant data, base join index definitions on `foreign key=primary key` predicates. Add either the primary key or the foreign key to the index definition, but not both, because the Optimizer has the intelligence to derive either from the other.

Adding Join Constraints That Facilitate Joining to Other Tables

Include join constraint columns that support joining to tables not defined in the join index. It makes no difference whether you assign these columns to the fixed part of the join index or to the repeating part.

For example, you might carry the join columns `c_nationkey` and `l_partkey` in the join index to facilitate joining the nation and parts tables to the join index.

When the join index is defined using an outer join, use the outer table join column rather than the inner table join column to enhance performance.

Specifying a Row-Partitioned or a Value-Ordered Sort Key for Range Queries

Sorting a join index by data values, as opposed to row hash values, is especially useful for range queries that involve the sort key. This means that defining a join index designed to support range queries using row-partitioning can significantly facilitate the performance of range queries.

Similarly, in some circumstances, you might obtain better performance by designing the join index without row partitioning, but using a value-ordered NUSI.

Note the following limitations:

- Value ordering is limited to a single numeric or DATE column with a maximum length of four bytes.
- The column you specify in the ORDER BY clause must be drawn from the set of fixed columns. You cannot order by a column from the set of repeating columns.

In the following example, the join index rows are hash-distributed across the AMPs on the primary index, `c_name`. Once assigned to an AMP, the join index rows are value-ordered using `c_custkey` as the sort key.

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
  FROM Orders LEFT JOIN Customer ON o_custkey = c_custkey
  ORDER BY o_custkey
  PRIMARY INDEX (c_name);
```

Join Index Benefits and Costs

Before you create a join index to optimize one or more of your frequently repeated queries, you should consider the following issues carefully:

- Cost of disk resources required to store a join index
- Cost of creating a join index
- Cost of maintaining a join index

- Benefits of reduced performance times for the queries the join index is designed to facilitate.

Cost

Cost is expressed as follows:

- Disk resources: Cost is expressed in bytes (space or size). Join indexes are space-intensive because they require separate tables, often of fairly high cardinality, for storage.
- Creation and maintenance: cost is expressed in time (elapsed time and CPU path per transaction).

Elapsed time is a fairly simple measure, but it is not comparable across different system configurations, nor is it independent of other workloads running on a system. It has the advantages of conceptual simplicity and ease of measurement. With respect to join index creation, it is a measure of how long it takes to create the index. With respect to maintenance, it is a measure of how long it takes to update a join index table beyond the time required to perform the update of the base table. In this sense, the term *update* refers generically to the data manipulation operations insert, update, and delete performed by any of the following SQL statements:

- DELETE
- INSERT
- MERGE
- UPDATE

CPU path per transaction is a sophisticated measure that is comparable across different system configurations. It has the disadvantages of being both difficult to measure without the proper tools and difficult to understand conceptually.

Whether creation and maintenance cost is measured as elapsed time or as CPU path per transaction, the same proportional relationship holds: the smaller the number, the better.

The total cost for a join index is the sum of its creation and maintenance costs.

Join Index Costs Summary

Creation and maintenance costs for join indexes can be a resource burden because of their processing overhead. In the case of join index maintenance, the burden is an ongoing process that lasts for the life of the index.

Costs vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- Maintenance overhead for in-place aggregate join indexes ranges from 1.0 to 2.9.
- Maintenance overhead for aggregate join indexes that redistribute rows ranges from 2.6 to 9.5.
- Maintenance overhead for simple join indexes ranges from 1.0 to 23.1, increasing as the number of hits per data block increases.
- Maintenance overhead for aggregate join indexes decreases as the number of hits per data block increases because of the efficiencies of block-at-a-time operations in the file system.

- Maintenance overhead for insert operations is significantly less than the maintenance overhead required for update and delete operations.

Benefit and Benefit Percentage

Benefit is a measure of the difference in elapsed time for the same query performed with and without a join index.

Benefit percentage is a normalized expression of benefit. It is defined as the product of benefit and 100 divided by the elapsed query time without a join index.

The larger the number, the better.

Payback

Payback is an expression of the number of queries that must be run to achieve a break even point for the total cost of the join index. Its value is derived from dividing total costs by benefit.

The smaller the number, the better.

Cost of Disk Resources

A join index table has the same properties as a base table except that it cannot be queried directly.

Because it is essentially a base table, the cardinality of a simple join index is generally of the same order as its component join table with the highest cardinality, with adjustments being necessary for row compression and other miscellaneous issues. The cardinality of an aggregate join index is typically much smaller than that of any of its component join tables.

The degree of either type of join index table is entirely dependent on how it is defined.

Irrespective of any other factors, join index tables extract a cost in terms of increased disk space requirements. If a join index table is fallback-protected, then it exerts twice again the burden in disk space.

The disk resources required for a join index are described in [Join Index Storage](#).

Cost of Join Index Creation

The cost of creating a join index is described in terms of time: how long does it take to create the index? Equally important in many situations is the question of how many times a query for which a join index was designed must be performed to offset the creation time by recovering response time.

Different types of join index require a longer or shorter time to create.

If your processing environment requires that join indexes be created and dropped on a regular basis, the cost of creating the index becomes a critical factor that must be considered carefully in determining its worth.

Because of this creation overhead, you should evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its creation cost if you must drop and create your join indexes frequently. If the gross effect of an index is negative with respect to overall system performance, you might not want to create it even if it enhances the performance of particular queries.

Unfortunately, it is not possible to predict join index creation times with any degree of accuracy because there are too many significant variables involved to be able to produce a useful predictive model. This

multivariate complexity also prevents the derivation of a useful predictive model for maintenance overhead or even for query time reductions.

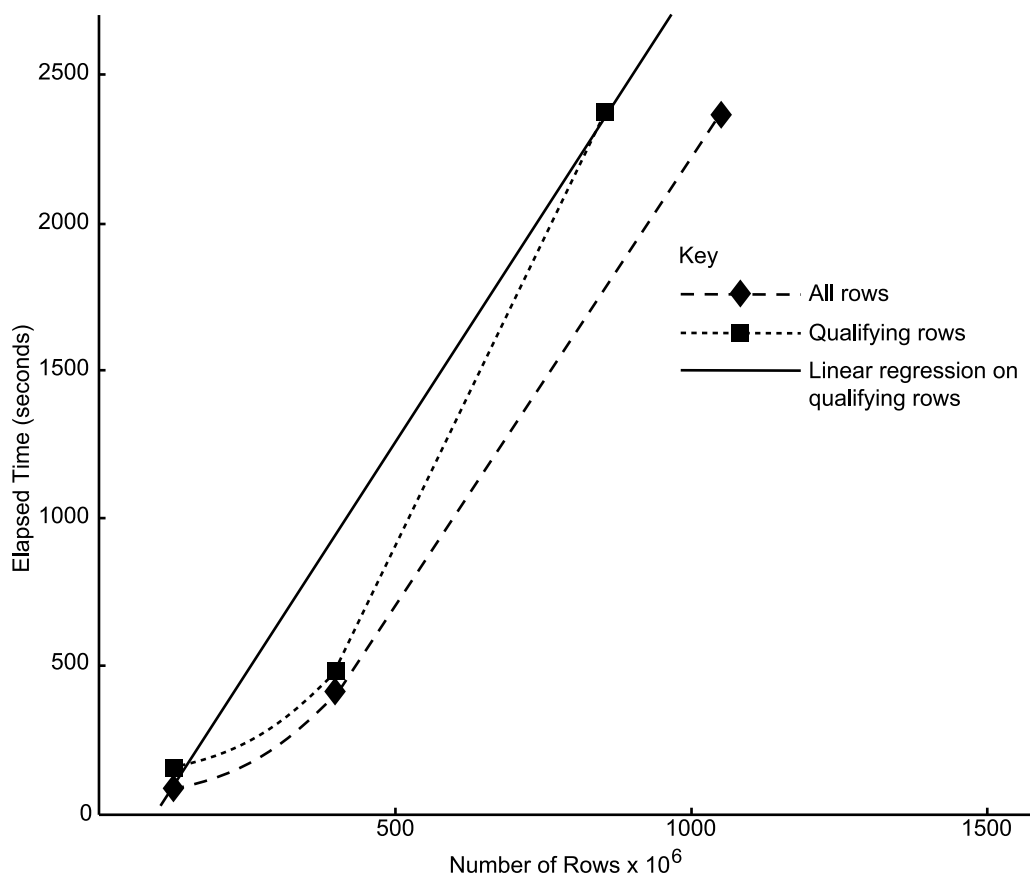
Example of Join Index Creation

Consider the following example. Empirical testing indicates that one of the simplest relationships is that between the time required to create an in-place aggregate join index and the number of rows in the base tables. Even this simple relationship is complicated by the fact that the resulting creation time is likely to be more closely related to the number of rows that qualify for the join rather than the total number of rows.

The following table indicates a measured relationship between the elapsed creation time for a join index and the number of qualified rows.

Elapsed Time (seconds)	Number of Qualified Rows x 10 ⁶
133	90
415	300
2,344	900

These numbers, along with the total number of rows in each test and a least squares linear regression of number of qualifying rows on elapsed time, are graphed in the following figure:



Creation and Elapsed Query Times for Different Join Indexes

The following tables list the creation and elapsed query times for several different kinds of join indexes.

Note that when the definition of a join index can be covered by an existing join index, the existing index can be used to create the join index.

In the following table, when calculating the values for the Query Ratio and Benefit columns, query times of less than 1 second were rounded up to 1.0 seconds.

Approximate Creation Costs and Query Benefits in Terms of Elapsed Query Time						
Join Index Type	Creation Cost (seconds)	Elapsed Query Time (seconds)		Query Ratio (seconds)	Benefit (seconds)	Benefit Percent
		Without Join Index	With Join Index			
Single-table aggregate	51	41	< 1	41.00	40	98
2-table in-place aggregate	135	107	< 1	107.00	106	99
4-table in-place aggregate	279	243	< 1	243.00	242	100
Single-table simple	199	80	63	1.27	17	21
2-table in-place	190	116	78	1.49	38	33
2-table in-place outer join	231	147	98	1.50	49	33
2-table foreign aggregate	232	230	< 1	230.00	229	100
2-table ad hoc aggregate	397	357	< 1	357.00	356	100

In the following table, the figure <<1 means the CPU path per row time was too short to measure. When calculating the values for the Benefit column, query times of less than 1 second were rounded up to 1.0 seconds. The Benefit value is based on elapsed query time rather than CPU path per row time.

Creation Costs and Query Benefits in Terms of CPU Path Per Row					
Join Index Type	Creation Cost (μseconds)	Query Time CPU Path Per Row (μseconds)		Benefit	Benefit Percentage
		Without Join Index	With Join Index		
Single-table aggregate	9	7	<< 1	40	98
2-table in-place aggregate	7	6	<< 1	106	99
4-table in-place aggregate	10	8	<< 1	242	100
Single-table simple	33	14	11	17	21
2-table in-place	14	10	7	38	33
2-table in-place outer join	16	11	7	49	33
2-table foreign aggregate	19	19	<< 1	229	100
2-table ad hoc aggregate	34	30	<< 1	356	100

You can use these figures, which are derived from tests on a two node system, to scale an approximately linear adjustment for your own table sizes and configuration.

Cost of Join Index Maintenance

The largest resource burden for any join index is incurred by its maintenance: inserting, updating, and deleting rows in the join index table. Similar to creation costs, various types of join indexes incur very different maintenance costs.

Maintenance Costs of Join Indexes

Join indexes can be expensive to maintain. For many customers, the most important factor in the decision to use a join index is likely to be how much it costs to maintain.

Each time a join-indexed base table column is updated, the corresponding join index table column must also be updated. Each time a new row is added to or an existing row is deleted from a join-indexed base table, the corresponding join index table rows must also be inserted or deleted.

Because of this maintenance overhead, you should always carefully evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its cost to create and maintain (see [Cost/Benefit Analysis for Join Indexes](#)).

Maintenance Cost Optimizations Based on Foreign Key-Primary Key Joins

You should consider the added cost of join index maintenance carefully when you are designing the indexes for your data warehouse to ensure that the minimum number of join indexes can be called upon by the Optimizer to cover the maximum number of queries. Designing with foreign key-primary key joins allows you to make these optimizations.

Whenever a base table column set that is shared with a join index is updated or deleted, or when a new row is inserted into the base table, the system generates extra steps to maintain the base table and join index concurrently. If the base table is specified as part of an outer join in the join index definition, the steps can be more complex because maintenance might be needed for both matched and unmatched row sets.

However, when the join columns have a foreign key-primary key relationship, the system treats inner and outer joins alike (see [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)).

Maintenance Cost Optimizations for DELETE ALL Operations

A *fastpath* optimization is one that can be performed faster if certain conditions are met. For example, in some circumstances DELETE and INSERT operations can be performed faster if they can avoid reading the data blocks and avoid transient journaling.

Vantage uses both fastpath and deferred fastpath row partition DELETE operations for the following cases:

- Deferred row partition deletion on a row-partitioned base table when a join index defined on the base table is not row-partitioned
- Deferred partition deletion on a row-partitioned join index that is defined on a table
- Deferred partition deletion on both a row-partitioned join index and its row-partitioned base table

In this case, Vantage performs the fastpath row partition deletion operations on the join index and the base table independently.

Vantage can perform fastpath DELETE ALL operations, but not deferred row partition deletion operations, for the following cases:

- If the deleted table has a conditional DELETE with predicates and it covers the entire join index, the join index is eligible for a fast path DELETE.
- All single-table join indexes.
- A multitable join index when the join between the tables is either an inner join or the table being deleted is the outer table in the join.
- An implicit transaction with a single-statement DELETE ALL *table_name* when the table has a join index defined on it.
- An implicit transaction with a multistatement request.
- An ANSI/ISO session mode transaction with a multistatement request.
- A Teradata session mode transaction with a multistatement request.

Types of Join Index Examined for This Analysis

Different types of join indexes incur different costs of maintenance. For the analyses provided here, several different types of join index were used. The types are far from exhaustive, but they provide a fairly representative range of data that you can use to extrapolate roughly how much benefit a join index that uses a particular type of join is likely to provide.

The following table provides a list of the various types of joins defined for the join indexes used in this study. When you think of types of joins, you probably think of join processing types like merge join, nested join, product join, and so on.

The types of joins defined here (in-place, foreign, and ad hoc) are unrelated to those join processing types. The types of joins and the types of join indexes go hand in hand. In both cases, it is the number of tables that are being redistributed that determines the type of join or join index being described.

Each join type can be used with either a simple or an aggregate join index.

Join Type	Definition
In-place	<p>The joined tables have a common primary index and are joined on that column set. A common example is a logical entity-subentity relationship as seen in an Employee - Employee_Phone relationship, where both tables have Employee Number as a common primary index.</p> <p>As a result, rows to be joined are always on a common AMP and do not have to be redistributed. This is the least expensive join of the three types examined.</p>
Foreign	<p>The tables are joined on a primary key - foreign key column set relationship, where the primary key column set is also the primary index column set. The primary index of the other join table is typically a foreign key in the first table.</p> <p>A common example is a join between an Employee table and a Department table, where the join is on the common Employee Number column, which is the primary index for the Employee table, but is only a foreign key in the Department table.</p> <p>As a result, the rows to be joined must be redistributed from the AMP having the Department table rows to the AMP having the Employee table rows.</p> <p>This join is more expensive than the homogeneous primary index join, but less expensive than the ad hoc join.</p>
Ad hoc	<p>The tables are joined on a column set that is not a primary index in either table. As a result, rows from both tables must be redistributed to make the join.</p> <p>This join is the most expensive of the three types examined.</p>

Maintenance Cost as Function of Hits for Each Data Block

The number of hits per block is a measure of how many rows in a data block are accessed (inserted, deleted, or updated) during an operation on a table. Generally speaking, the greater the number of hits per block, the better the performance provided the hits can be combined into one update of the data block, as would be the case, for example, with an INSERT ... SELECT involving a set of updates containing multiple rows.

If a large number of the data blocks for a table have become significantly smaller than half the maximum size for the defined maximum data block size, an appropriate specification for the MERGEBLOCKRATIO option of the ALTER TABLE and CREATE TABLE statements can enable the file system to merge up to 8 data blocks into a single larger data block. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

On the other hand, if each hit of a data block involves a separate read operation, with hits occurring semi-randomly, performance is worse.

Data Processing Scenarios for the Hit Rates Studied

The following table provides example data processing scenarios that correspond with the hit rates studied.

Number of Hits per Data Block	Percentage of Rows Touched	Scenario
1	0.2	Update a table that contains 500 days of data (roughly 1.5 years) with one typical day of data.
5	1.0	Insert into a table that contains 104 weeks of data (2 years) one typical week of data.
20	4.0	Delete from a table that contains 104 weeks of data (2 years) one typical month (4 weeks) of data.

Standard Test Procedure

The standard procedure for these tests was to make any maintenance changes to the left table in the join.

Maintenance Costs In Terms of Elapsed Time

Elapsed times increase as a function of increased hits per data block because more rows are touched. At the same time, the CPU path per row times decrease because the elapsed times increase at a lesser rate than the increase in the number of rows touched.

Maintenance Costs In Terms of CPU Path Length Per Transaction

Suppose you report the same information in terms of CPU path length per transaction. For this data, the term *transaction* equates to *qualifying row*. This number is a measure of the amount of CPU time a transaction requires; roughly, the number of instructions performed per transaction.

CPU path per transaction is a better way to compare various manipulations than elapsed time for two reasons:

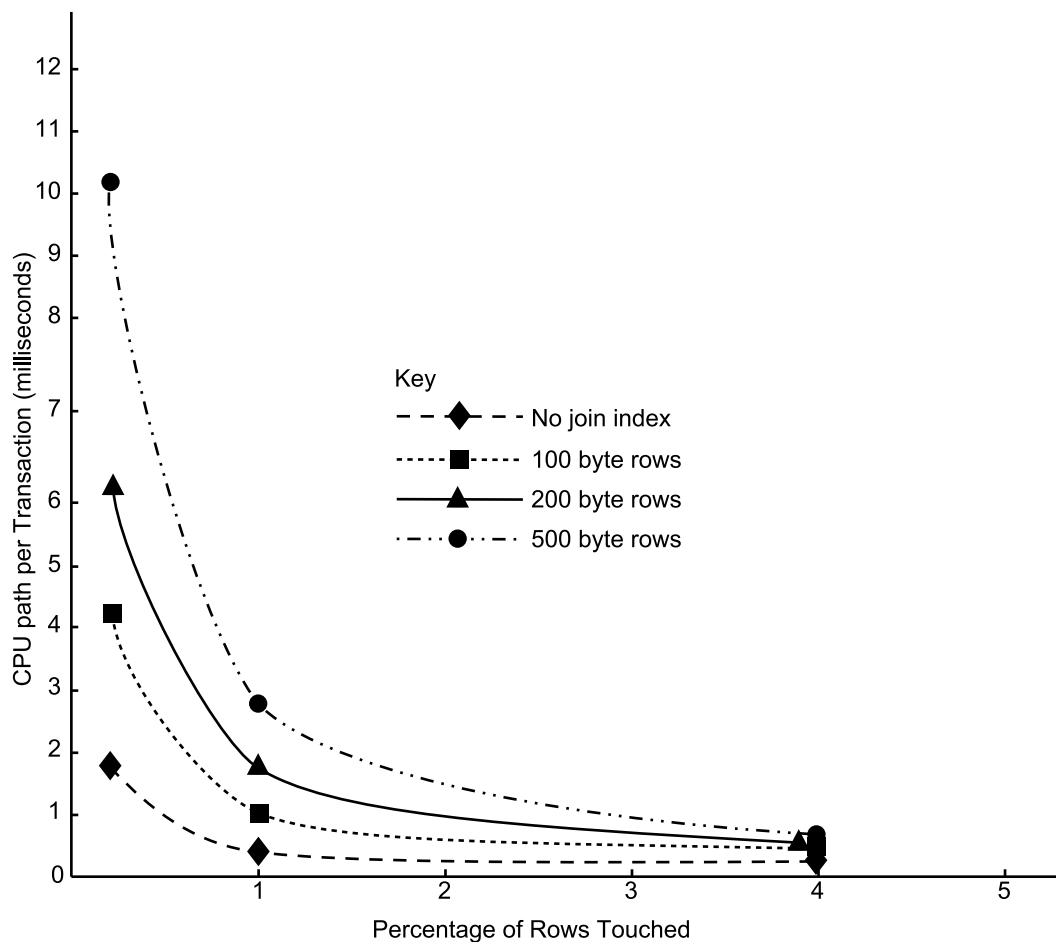
- CPU path per transaction figures include both CPU utilization information and elapsed time.
- Because the measure is normalized, it factors out configuration-specific variables such as number of CPUs per node and number of nodes, making the numbers comparable across all configurations.

Maintenance Cost as a Function of Row Size

The size of join index rows is another important factor to consider when evaluating the maintenance overhead of a join index. Given a constant disk space, as the size of the rows increases, the number of rows decreases, while the number of data blocks remains the same. The result is that the number of hits per data block decreases.

Maintenance Costs in Terms of CPU Path Per Transaction

CPU path per transaction times for updates increase steadily as a function of increased row size, while the number of transactions, or qualified rows, decreases drastically. This can be seen in the following graph, where CPU path per transaction is presented as a function of percentage of rows touched during an update operation.



Maintenance Costs in Terms of Other Measures

The following table indicates the trends in performance cost of join index maintenance as a function of increasing row size in terms of various performance measures:

Performance Measure	Effect as a Function of Increasing Row Size
Elapsed time to insert, delete, or update join index rows.	Decreases
I/Os per insert, delete, or update transaction	Increases
Number of point-to-point BYNET insert, delete, or update transactions	Increases

Summary

Summarizing these results, each join index maintenance transaction performs more work as a function of increased row size as measured by the following list of variables.

- CPU path per transaction
- I/Os per transaction
- BYNET transactions per transaction

Join Index Maintenance Cost as a Function of Insert Method

For myriad reasons, some methods of inserting rows into tables are much more efficient than others. Four different methods of insertion are examined here for their efficiency in lowering the maintenance cost of join indexes. The following methods are reported.

- Case 1: FastLoad and INSERT ... SELECT the rows
- Case 2: Drop the join index, insert the rows into the base table, recreate the join index
- Case 3: Teradata Parallel Data Pump the rows into the base table
- Case 4: Insert the rows into the base table using SQL

Case 1: INSERT ... SELECT

This method uses the following procedure.

1. FastLoad the rows into an empty table.
2. INSERT ... SELECT the rows from the freshly loaded table into the table that has a join index defined.

This procedure uses block-at-a-time optimization and is the fastest of the methods examined.

Case 2: Dropping a Join Index and Recreating It After Inserting Rows Into Its Base Table

This method uses the following procedure:

1. Drop the join index.
2. Insert the rows into the base table using any method.
3. Recreate the join index.

The elapsed time for this method averages to about 1.3 times the elapsed time measured for the method provided in Case 1.

Case 3: Teradata Parallel Data Pump

This method uses the Teradata Parallel Data Pump utility to insert rows into the base table with a join index. The Teradata Parallel Data Pump utility has several procedural advantages that make it an attractive option, including restartability and resource throttleability.

Because the Teradata Parallel Data Pump utility performs row-at-a-time operations, however, its performance for this application is not optimal. Measurements confirm that the Teradata Parallel Data Pump utility, when performed using a non-optimized run, is more than 200 times slower than the method of Case 1 with respect to elapsed time measurements. Optimization of the run using techniques like sorting input data in different orders can reduce this figure by an order of magnitude in many circumstances, bringing the cost differential down to a more reasonable 20:1 ratio for a 1 hit per data block situation. As the number of hits per data block increases, the cost differential increases because methods that use block-at-a-time methods become increasingly efficient.

Case 4: Single Row INSERT Request

This method uses the SQL INSERT statement to insert the rows into the base table. The data presented in [Maintenance Cost as Function of Hits for Each Data Block](#) confirms that this method is slightly better than the method presented in Case 2.

Comparative Elapsed Times to Insert

The following table indicates the comparative elapsed times required to insert the same number of 100 byte rows into a base table and join index table.

Insertion Method	Operation Performed	Elapsed Time (seconds)
Case 1:	FastLoad into empty table	26
	INSERT ... SELECT into base table with join index	99
	Total	125
Case 2:	Drop join index	0
	INSERT ... SELECT into base table without join index	59
	Recreate join index	104
	Total	163
Case 3:	Use the Teradata Parallel Data Pump utility to load rows into a base table with a join index	25,896

Summary Evaluation

The following table presents a summary evaluation of the insert methods examined in this topic.

Insertion Method	Evaluation
Case 1:	Greatest speed
Case 2:	Least advantageous method
Case 3:	<ul style="list-style-type: none"> • Relatively slow • Offers other advantages
Case 4:	<ul style="list-style-type: none"> • Simple to perform • Greatest efficiency for a small number of rows

Generalizations Derived From These Tests

The following generalizations are made from these test results:

- Maintenance costs for aggregate join indexes are much lower than maintenance costs for comparable simple join indexes.
- Maintenance costs for no join index are lower than maintenance costs for foreign and ad hoc join indexes.

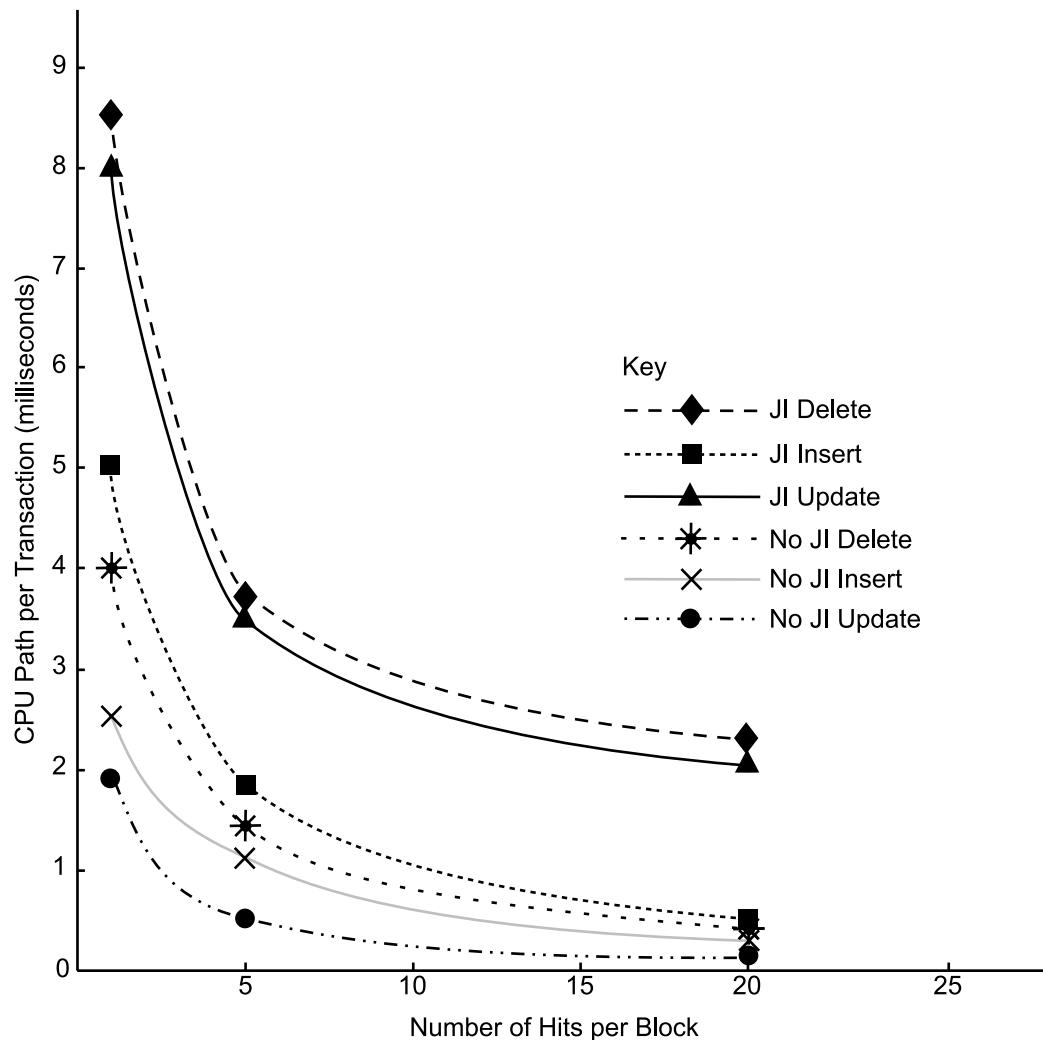
For example, maintenance costs for a 2-table in-place aggregate join index are 1.1 to 2.5 times greater than maintenance costs computed for just the base table without a join index.

- Maintenance costs vary with the type of join index defined.

For example, with an in-place aggregate join index, the higher the number of hits per data block, the less the overhead incurred. This effect is marginal for inserts, for which maintenance costs are already minimal.

On the other hand, for an in-place simple join index at 1 hit per data block, deletes cost 3.4 times more and updates cost 4.5 times more than the case where no join index is defined.

In contrast to the in-place aggregate join index, maintenance costs worsen as the number of hits per data block increase, as indicated by the following graph of CPU path per transaction as a function of number of hits per data block.



In the case of an in-place simple join index, inserts cost less than 1.4 times the maintenance required when no join index is defined.

In all cases, inserts are never more than four times more expensive than the maintenance cost when no join index is defined.

Cost/Benefit Analysis of Join Indexes

Any query performs faster when it uses a join index than it does without a join index. The issue that must be examined is whether or not the benefit of the decreased response times provided by a join index offset the costs of its creation, maintenance, and storage.

The relative costs and benefits of defining a join index are described in [Cost/Benefit Analysis for Join Indexes](#).

Cost/Benefit Analysis for Join Indexes

This topic introduces the concepts of cost/benefit analyses for various types of join indexes. Several different computations are described to support these analyses including the following metrics.

- Cost
- Benefit
- Benefit percentage
- Payback
- Query ratio

Join Index Benefits Summary

The benefits of join indexes vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- All queries that require join processing benefit from join indexes; often dramatically, and sometimes spectacularly.
- In all tests performed, aggregate join indexes strongly outperform simple join indexes, both with respect to their performance and with respect to their maintenance burden. Queries using an aggregate join index frequently run hundreds of times faster than the same queries against the same tables when no join index is defined.
- The benefits of simple join indexes are typically more modest. Queries using a simple join index typically run from 1.3 to 4.9 times faster than the same queries against the same tables when no join index is defined.
- In-place join indexes handily outperform any join indexes that redistribute rows.

Computing the Benefits of Join Indexes

This topic describes a measure you can use to calculate the benefit of a join index.

Begin the calculation by computing some preliminary measures, as described in the following set of equations.

Define the benefit of a join index as follows:

$$\text{Benefit} = \text{ET}_{wo} - \text{ET}_w$$

where:

Syntax element ...	Specifies elapsed query time ...
ET_{wo}	without a join index defined for the query.
ET_w	with a join index defined for the query.

Benefit is a simple measure that measures the advantage of a query in terms of the difference between its elapsed completion time when a join index is not defined and its elapsed completion time when a join index is defined.

Define the benefit percentage as follows:

$$\text{Benefit Percentage} = (ET_{\text{wo}} - ET_{\text{w}}) \times \frac{100}{ET_{\text{wo}}}$$

The benefit percentage for a query is just a normalized form of the raw benefit. It provides an easily understood measure of the reduction in processing time gained by creating the join index.

Rearrange terms to determine the elapsed time to process a query with the join index defined. This measure is necessary to determine if the cost of creating the join index exceeds its usefulness in reducing response time.

$$ET_{\text{w}} = ET_{\text{wo}} - \text{Benefit Percentage} \times \frac{ET_{\text{wo}}}{100}$$

Computing the Query Ratio

Define the query ratio as follows:

$$\text{Query Ratio} = \frac{\text{Elapsed query time without join index}}{\text{Elapsed query time with join index}}$$

The query ratio is a normalized measure of how much faster a query executes with a join index than without a join index. For example, consulting the table about [Creation and Elapsed Query Times for Different Join Indexes](#), you find a column of calculated query ratios.

You interpret these ratios as follows. Consider the query ratio for the creation cost of a single-table aggregate join index. The reported query value for this join index is 41. This means that the test query runs 41 times faster with the single-table aggregate join index defined than it does without it.

Computing the Payback Factor

This topic explains how to calculate the payback factor for a join index. Note that the term Costs is something you must measure. It is the time required to compute the join index being investigated.

$$\text{Payback Factor} = \text{Costs} \div \text{Benefit}$$

where:

Syntax element ...	Specifies ...
Costs	the processing time required to create the join index.

Because the term *payback factor* represents a number of queries, it is always rounded up to represent a whole number. The smaller the payback factor, the sooner benefits accrue from using the join index.

Example of Computing a Payback Factor for a Simple Join Index

Suppose you have a frequently performed, non-aggregate join query that you think might benefit from a join index. You decide to test the join index and collect the following data.

Parameter	Measured Value (seconds)
ET _{wo}	416
ET _w	226
Cost	1055

What is the payback factor for creating this join index?

Benefit = 416 – 226 = 190 seconds

Benefit Percentage = $190 \times (100 \div 416) = 46\%$

The interpretation of the benefit percentage is that the time to complete the query is cut nearly in half.

Payback factor = $1055 \div 190 = 6$

The interpretation of the payback factor is that the cost of creating the join index is recovered if the query is run six or more times.

Two More Brief Examples

The following examples work through the following join indexes.

- In-place simple join index
- Ad hoc aggregate join index

Follow the identical procedure to that used in the detailed example.

For the in-place simple join index, the calculation is as follows.

$$\text{Payback} = \frac{1900 + (2064 \times 4)}{380} = 27 \text{ queries}$$

To recover its cost in a 30-day period, this query must be run slightly more than once per business day.

For the ad hoc aggregate join index, the calculation is as follows.

$$\text{Payback} = \frac{3970 + (2510 \times 4)}{3560} = 4 \text{ queries}$$

To recover its cost in a 30-day period, this query must be run once per week.

Join Index Types

You can define several types of join index, each having its unique role in enhancing the performance of your database queries without denormalizing the base tables that support your ongoing ad hoc data warehouse activities. Because join index tables cannot be accessed directly by users and are not part of the logical design for a database, they can be used to create persistent prejoin and summary tables without removing or otherwise lessening the ability of your databases to support a wide range of ad hoc queries.

You can also create views whose SELECT definitions are identical to those of a join index, then whenever a user accesses data using that view, the similarly-defined join index should cover the request, and thereby be selected by the Optimizer for the query plan it develops (unless it discovers and chooses a plan that is less costly).

Note that any join index type can be defined with a row-partitioned primary index as long as it is not also row-compressed.

Following is a summary of the join index types and their common uses.

Join Index Type	Description
Simple	A join index table defined without aggregation.
Single table	<p>Several potential applications exist.</p> <ul style="list-style-type: none"> A column subset of a very large base table defined with a NUPI defined on a join key that causes its rows to be hashed to the same AMP as another very large table with a primary index defined on that same join key and to which it is frequently joined. Useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs. Subentities are typically small tables, while major entity tables are typically quite large. The index can emulate vertical partitioning of the base table by selecting a small subset of the most frequently accessed columns of a very wide table. Several possibilities exist for the vertical partitioning solution: <ul style="list-style-type: none"> The base table and the join index can have the same primary index. The base table and the join index can have different primary indexes. This option is particularly useful when frequent requests against the base table specify predicates on non-primary index columns. The index can be defined to contain all the columns defined for its base table, but with a different primary index. Another variable to consider is alternate orderings of the index: <ul style="list-style-type: none"> Hash ordering. Value ordering.

Join Index Type	Description
	<ul style="list-style-type: none"> A final valid, though pointless, option is to define the single-table join index over all the columns of its base table and also to define its primary index on the same column set as the base table. <p>All this does is create a mirror image of the base table that not only will never be used by the Optimizer for its query plans, but which also adds a great deal of useless maintenance overhead to the system.</p> <p>See Single-Table Join Indexes for additional information.</p>
Multitable	<p>A column subset of two or more major tables that are frequently joined defined as a prejoin of those tables.</p> <p>The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries rather than specifying that its underlying base tables be searched and joined dynamically.</p>
Aggregate	A join index table defined with aggregation on one or more of its columns.
Single-table aggregate	<p>A column subset of a base table defined with additional columns that are aggregate summaries of base table columns.</p> <ul style="list-style-type: none"> If the summary table is to be used simply to maintain aggregates for an overlying base table without denormalizing the database, then its NUPI need not be defined on a join key column set. If the summary table is to be used to maintain aggregates for an overlying base table and to be joined frequently with another very large table, then its NUPI should be defined on a join key column set that hashes its rows to the same AMPs as the base table primary index.
Multitable aggregate	<p>A column subset, including aggregates defined for one or more columns, of two or more major tables that are frequently joined defined as a prejoin of those tables.</p> <p>The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries that also compute aggregates rather than specifying that its underlying base tables be searched, aggregated, and joined dynamically.</p>
Sparse	<p>Any join index that limits its rows to those satisfying a constant condition in the WHERE clause of its definition.</p> <p>A sparse join index permits the database designer to limit the rows in the index to a tightly filtered subset of the rows in the component base table set. A common use of this feature might be to restrict the population of a join index to only those rows that are frequently accessed by a commonly performed query or set of queries.</p>
Column-partitioned	<p>Useful in support of DML requests that access a variable, selective, small subset of the columns (as specified in predicates or projection lists) and rows of a column-partitioned table. Also useful in support of RowID joins between a column-partitioned table and another table. Column-partitioned join indexes have the following restrictions:</p> <ul style="list-style-type: none"> Must be a single-table join index Cannot compute aggregates Cannot be row-compressed Cannot have value-ordering (but can have row partitioning).

Simple Join Indexes

The primary function of a simple join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently performed join operation. The simple join index permits you to define a permanent prejoin table without violating the normalization of the database schema. Simple join indexes are also referred to as multitable join indexes.

Simple Join Index Example

For example, suppose that a common task is to look up customer orders by customer number and date. You might create a join index like the following, linking the customer table, the order table and the order detail table:

```
CREATE JOIN INDEX cust_ord2
AS SELECT cust.customerid,cust.loc,ord.ordid,item,qty,odate
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid;
```

While you might never issue a query that completely joined these three tables, the key benefit of this join index is its versatility. For example, a query that only looks at the customers for a single state, like the following, can still use the cust_ord2 join index rather than accessing its underlying base tables.

```
SELECT cust.customerid, ord.ordid, item, qty
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid
AND cust.loc = 'WI';
```

Simple Join Indexes and Partial Query Covering

It is also possible for a join index to be used to partially cover a query to improve query performance. For example, if you wanted to count the number of orders made by customers in the European region during October, you might use the following query:

```
SELECT cust.customerid,COUNT(ord.ordid)
FROM cust, ord, orditm, location
WHERE ord.ordid = orditm.ordid
AND cust.customerid = ord.customerid
AND cust.loc = location.loc
AND location.region = 'EUROPE'
AND EXTRACT(MONTH, ord.orddate) = 10
GROUP BY cust.customerid;
```

In this example, the query includes the location table which is not included in the join index. Vantage can still use the join index to partially cover the query by joining the contents of the join index with the location table.

See [Partial Query Coverage](#) for more information about partial query coverage.

Defining a Simple Join Index on a Binary Join Result

Table Definitions

Suppose you define the following customer and orders tables.

```
CREATE TABLE customer (
  c_custkey    INTEGER NOT NULL,
  c_name       CHARACTER(26) NOT NULL,
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
PRIMARY INDEX(c_custkey);

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_date        DATE FORMAT 'yyyy-mm-dd',
  o_status      CHARACTER(1),
  o_custkey     INTEGER,
  o_totalprice  DECIMAL(13,2),
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
  o_comment     VARCHAR(79))
UNIQUE PRIMARY INDEX(o_orderkey);
```

Example Query Request

Consider the following SELECT request against these tables.

```
SELECT o_custkey, c_name, o_status, o_date, o_comment
FROM orders, customer
WHERE o_custkey=c_custkey;
```

The next few topics examine the query plan for this SELECT request: first without and then with a join index.

Query Plan: No Join Index Defined

Without a defined join index, the execution plan for this query would typically redistribute the orders table into a spool, sort the spool on o_custkey, and then perform a merge join between the spool and the customer table.

Query Plan: Join Index Defined

Now consider the execution plan for this same query when the following join index has been defined:

```
CREATE JOIN INDEX OrdCustIdx AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM orders, customer
WHERE o_custkey=c_custkey;
```

With this join index defined, the execution plan for the query specifies a simple scan of the join index without accessing any of the underlying base tables and without having to join them on the predicate WHERE o_custkey = c_custkey.

In the join index defined for this example, (o_custkey, c_name) is the specified fixed part of the index and (o_status, o_date, o_comment) is the repeated portion. Therefore, assume the following specimen base table entries (where the ? character indicates a null).

Customer

CustKey	Name	Address
100	Robert	San Diego
101	Ann	Palo Alto
102	Don	El Segundo

Orders

OrderKey	Date	Status	CustKey	Comment
5000	2004-10-01	S	102	rush order
5001	2004-10-01	S	100	big order
5002	2004-10-03	D	102	delayed
5003	2004-10-05	U	?	unknown customer
5004	2004-10-05	S	100	credit

You cannot collect statistics on a complex expression from a base table. If your applications frequently run queries that specify complex expressions in their predicates, you should consider creating a single-table join index that specifies a matching complex expression in its select list or column list, respectively. When

Vantage creates the index, it transforms the complex expression into a simple index column on which you can collect statistics.

If the complex expression specified by the index is a term that matches a predicate condition for a query made against the base table the index is defined on, statistics collected on the index expression can be mapped to the base table so the Optimizer can use them to make more accurate single-table cardinality estimates.

Materialized Join Index

The materialized logical join index rows are the following:

OrdCustIdx

Fixed Part		Repeated Part		
CustKey	Name	Status	Date	Comment
100	Robert	S	2004-10-01	big order
		S	2004-10-05	credit
101	Ann	P	2004-10-08	discount
102	Don	S	2004-10-01	rush order
		D	2004-10-03	delayed

Note that the information for the null customer is not included in this join index because it was defined using an inner join.

Note:

Join indexes are not limited to binary joins: like any other join, they can be defined on joins involving as many as 128 tables.

Defining and Using a Simple Join Index With an n -way Join Result

The following example shows the creation of a join index defined with an n -way join result and then shows how the Optimizer uses the join index to process a query on the base tables for which it is defined.

Join Index Definition

The following statement defines a join index with a three-table join using both natural and outer joins.

```
CREATE JOIN INDEX cust_order_join_line AS
  SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
         (l_partkey, l_quantity, l_extendedprice, l_shipdate)
  FROM (lineitem LEFT JOIN orders ON l_orderkey = o_orderkey)
```

```

INNER JOIN customer ON o_custkey = c_custkey
PRIMARY INDEX (l_orderkey);
*** Index has been created.
*** Total elapsed time was 20 seconds.

```

EXPLAIN for Query with Complicated Predicates

The following EXPLAIN shows how the Optimizer might use the join index for a query that accesses all three of the base tables defined in the index.

```

EXPLAIN SELECT l_orderkey, o_orderdate, o_totalprice, l_partkey,
              l_quantity, l_extendedprice, l_shipdate
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND    o_custkey = c_custkey
AND    c_nationkey = 10;
*** Help information returned. 16 rows.
*** Total elapsed time was 1 second.
Explanation
-----
1) First, we lock LOUISB.cust_order_join_line for read on a
   reserved Row Hash to prevent a global deadlock.
2) Next, we do an all-AMPs RETRIEVE step from join index table
   LOUISB.cust_order_join_line by way of an all-rows scan with a
   condition of ("LOUISB.cust_order_join_line.c_nationkey = 10") into
   Spool 1, which is built locally on the AMPs. The input table will
   not be cached in memory, but it is eligible for synchronized
   scanning. The result spool file will not be cached in memory.
   The size of Spool 1 is estimated to be 200 rows. The estimated time
   for this step is 3 minutes and 57 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.

```

Single-Table Join Indexes

You can define a simple join index on a single table. A single-table join index is a database object created using the CREATE JOIN INDEX statement, but specifying only one table in its FROM clause. This permits you to hash some or all of the columns of a large replicated base table on a foreign key that hashes rows to the same AMP as another large table. In some situations, this is more high-performing than building a multitable join index on the same columns. In effect, you are redistributing an entire base table or a frequently accessed subset of base table columns using a join index when you do this. The main advantage comes from less under-the-covers update maintenance on the single-table form of the index.

Single-table join indexes are the only type of join index that can be defined with a unique primary index.

The term *single-table join index* might seem to be a contradiction because there are no joins in a single-table join index. However, the Optimizer can use single-table join indexes to facilitate joins. The single-table join index came about because an observant software architect had the insight that it was possible to use the join index mechanism with a single table to horizontally partition all or a subset of a very large base table as a join index on a different primary index than that used by the original base table in order to hash its rows to the same AMPs as another very large base table that with which it was frequently joined. In this respect, a single-table join index is essentially a hashed NUSI.

Because of the way the rows of a column-partitioned join index are distributed to the AMPs, this advantage does not generalize to them or by the Fast Load utility. However, column-partitioned join indexes are useful as an alternative method to partition a base table in an entirely different way when such an option provides an appropriate choice for the Optimizer to consider for some queries.

This application is analogous to how NUPIs are often used in database design to hash the base table rows of a minor entity to the same AMP as rows from another table they are likely to be joined with in a well known query workload (see [Nonunique Primary Indexes](#)), though you cannot explicitly specify a join to a join index in a DML request. Instead, the Optimizer must determine if joining base table rows with join index rows would be less costly than other methods.

Functions of Single-Table Join Indexes

Even though each single-table join index you create partly or entirely replicates its base table, you cannot query or update them directly just as you cannot directly query or update any other join index.

When you have an application for which join queries against a base table would benefit from replicating some or all of its columns in a different table hashed on the join key (usually the primary index of the table to which it is to be joined) rather than the primary index of the original base table, then you should consider creating one or more single-table join indexes on that table.

For example, you might want to create a single-table join index to avoid redistributing a large base table or to avoid the sometimes prohibitive storage requirements of a multitable join index. A single-table join index might be useful for commonly made joins having low predicate selectivity but high join selectivity, for example.

This strategy substitutes the join index for the underlying base table and defines a primary index that ensures that rows containing only the columns to be joined are hashed to the same AMPs, eliminating the need to redistribute rows when the database manager joins the tables.

As another example, suppose you have a primary index defined on a major entity column that joins with many foreign key subentity columns. The cost of the maintenance required to update a multitable join index defined on this table is many times greater than the cost of maintaining the underlying base table.

The Optimizer can use unique single-table join indexes to access base table rows.

When you have a table with a large number of columns that is queried frequently, but only on a small subset of those columns, you can create either a hash index or a single-table join index to effectively partition the table vertically. Partitioning the rows of a table, as Vantage does to distribute rows to the AMPs, is often called *horizontal partitioning*. This is not what a single-table join index or hash index does. Instead, those indexes effectively partition tables on their columns, a method referred to as *vertical partitioning*. For example, for a table with 1,500 columns, only 25 of which are frequently queried, you could create a hash or single-table join index on those 25 frequently queried columns, which has the same effect as vertically partitioning the base table itself into two sets of columns: one set of 25 frequently queried columns and another set of 1,475 infrequently queried columns. Note that neither horizontal partitioning nor vertical partitioning is related in any way to how Vantage partitions the rows of a table having a partitioned primary index on an AMP, and that is why the terms horizontal partitioning and vertical partitioning are generally avoided in this document.

With a hash or single-table join index available that contains all of the frequently queried columns from the base table (and in the case of a join index, either the ROWID key word, the unique primary index of the base table, or a USI from the base table), the Optimizer can use that index to cover queries on that column subset, and then join to the base table to pick up any additional columns from the table that a query might specify in its select list.

You can also use single-table join indexes as a mechanism to collect statistics on complex expressions that are defined in their select list. The Optimizer can then either use mapping to exploit a matched expression that it finds in a non-matching predicate by mapping to the join index column statistics, or it can use matching when it detects identical predicates in both the join index definition and in a query made against the base table on which the join index is defined. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Column-Partitioned Single-Table Join Indexes

You can create column-partitioned single-table join indexes with the following restrictions.

- The index cannot compute aggregates.
- The index cannot be row-compressed.
- The index cannot have value ordering (but they can be row-partitioned).

Column-partitioned join indexes are designed to support requests that very selectively access a variable small subset of the columns and rows, either in predicates or as column projections.

The Optimizer can also use column-partitioned join indexes to support direct access to a column-partitioned table using a RowID join.

See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information about column partitioning and single-table join indexes.

Maintenance Costs of Single-Table Join Indexes

For a single-table join index, the maintenance cost is roughly double the cost of maintaining the base table.

When you design a schema, there are often some tables that are queried in such a way that for some frequently run workloads, the table is joined on one column, but for another important query, the table is joined on another column. The usual design solution is to distribute the rows of this table on the column that is most frequently used in a join. If there is more than one column, then a join index might be a good design choice. A join index can be used to redistribute the table on the secondary join attribute so that joins can be done without a redistribution step.

Join indexes can also be used to evaluate parameterized queries. For the Optimizer to use a join index in this situation, the query must also contain a non-parameterized condition in its WHERE clause that the join index covers.

For example, suppose you create the following base table and single-table join index:

```
CREATE TABLE tp1 (
  pid      INTEGER,
  name     VARCHAR(32),
```

```

address VARCHAR(32),
zipcode INTEGER);
CREATE JOIN INDEX tp1_ji AS
SELECT pid, name, zipcode
FROM tp1
WHERE zipcode >50000
AND   zipcode < 55000;

```

Parameterized Queries and Single-Table Join Indexes

The following parameterized query can use this join index because the Optimizer knows that the matching rows are contained in the index because the WHERE clause predicate in the query is a conjunction between the covered term zipcode and the parameterized term :N.

```

USING (N VARCHAR(32))
SELECT pid, name
FROM tp1
WHERE zipcode IN (54455, 53066)
AND   name = :N;

```

The explanation for this query looks like the following report:

Explanation

- ```

1) First, we lock CURT.TP1_JI for read on a
 reserved RowHash to prevent global deadlock.
2) Next, we do an all-AMPS RETRIEVE step from CURT.TP1_JI by way of an
 all-rows scan with a condition of ("((CURT.TP1_JI.zipcode = 53066)
 OR (CURT.TP1_JI.zipcode = 54455)) AND (CURT.TP1_JI.name = :N)")
 into Spool 1 (group_amps), which is built locally on the AMPS.
 The size of Spool 1 is estimated with no confidence to be 1 row (
 64 bytes). The estimated time for this step is 0.03 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPS involved
 in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
 statement 1. The total estimated time is 0.03 seconds.

```

Only prototyping can determine which is the better design for a given set of tables, applications, and hardware configuration.

## Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

The following list describes some of the alternative strategies to creating single-table join indexes:

- You can create a hash index.

For some applications, a hash index is a better choice than a single-table join index if only because of its simpler syntax; however, it might be unclear which defaults Vantage used to create the index. In nearly all cases you can, and should, create single-table join indexes that have identical effects on query workloads as the equivalent hash index. Also, multivalue compression can be carried over to join index but not for a hash index.

See [Hash Indexes](#) for more information.

- The design technique of assigning a NUPI to a subentity table that hashes related rows to the same AMPs as a related major entity is superficially similar to a single-table join index. The differences are as follows.
  - Cardinalities
 

The cardinalities of tables for which a single-table join index is defined are typically very similar to the base tables they are designed to be joined with, while those for major entity-subentity joins are typically very different, with the major entity typically having many more rows than the subentity.

The entity PI-subentity NUPI strategy is typically used when the subentity is a relatively small table in terms of its degree as well as its cardinality.

The single-table join index strategy is typically used when only a small subset of the columns from the base table from which the single-table join index is derived are frequently joined with the base table in question.
  - Specialization
 

When you create a single-table join index, the parent base table from which it is derived might have a different primary index, in which case its rows hash to different AMPs. The single-table join index is a denormalized, specialized database object defined for a specific purpose, while the parent base table is a normalized, more general database object. Both tables in an entity-subentity relationship remain normalized and generalized database objects.
- You can create a multitable join index that prejoins the entity attributes most likely to be joined in a query.
 

Updating a multitable join index can have a varying cost depending on which table in the multitable join is update, the indexes on this join index and base tables, and so on. In some cases, the update can be about the same as single-table, sometimes it can be very expensive if it requires an expensive join to be able to do the maintenance.

The upside of a standard multitable join index strategy is that, at least for the queries for which they are designed, Vantage does not have to perform any join processing because the required rows are already prejoined. The single-table join index can avoid a costly redistribution of table rows, but join processing is still required to respond to the query.
- You can create a denormalized prejoin base table.
 

Denormalization reduces the generality of the database for ad hoc queries and data mining operations as well as introducing various problematic update anomalies. Nevertheless, a relatively mild degree of denormalization is standard in physically implemented databases, and for some applications might be the only high-performing solution.

See [Single-Table Join Index](#) for an example of using a single-table join index.

## Single-Table Join Index

This example shows how a single-table join index can be used as a substitute for a standard join index to minimize update maintenance while at the same time making join processing more high-performing than it would otherwise be.

### Table Definitions

Suppose you have the following tables that you query frequently using join expressions, and both are very large.

| Table Name | Primary Index | Primary Index Type |
|------------|---------------|--------------------|
| LineItem   | OrderKey      | NUPI               |
| Part       | PartKey       | UPI                |

The table definitions are as follows.

```
CREATE TABLE LineItem (
 l_OrderKey INTEGER NOT NULL,
 l_PartKey INTEGER NOT NULL,
 l_SupplierKey INTEGER,
 l_LineNumber INTEGER,
 l_Quantity INTEGER NOT NULL,
 l_ExtendedPrice DECIMAL(13,2) NOT NULL,
 l_Discount DECIMAL(13,2),
 l_Tax DECIMAL(13,2),
 l_ReturnFlag CHARACTER(1),
 l_LineStatus CHARACTER(1),
 l_ShipDate DATE FORMAT 'yyyy-mm-dd',
 l_CommitDate DATE FORMAT 'yyyy-mm-dd',
 l_ReceiptDate DATE FORMAT 'yyyy-mm-dd',
 l_ShipInstruct VARCHAR(25),
 l_ShipMode VARCHAR(10),
 l_Comment VARCHAR(44))
PRIMARY INDEX (l_OrderKey);

CREATE TABLE part (
 p_PartKey INTEGER NOT NULL,
 p_PartDescription CHARACTER(26),
 p_SupplierNumber INTEGER)
UNIQUE PRIMARY INDEX (p_PartKey);
```

### Example Query Request

A frequently performed query on these tables might be the following:

```
SELECT l_PartKey, p_PartDescription, l_Quantity, l_SupplierKey
FROM LineItem, Part
WHERE l_PartKey=p_PartKey;
```

### Decision: Ordinary Join Index Versus Single-Table Join Index

You could create an ordinary join index on the LineItem and Part tables, but there is a high cost to keeping this join index updated because each update requires a costly minijoin operation, so the ordinary join index would have to be updated each time either a line item or a new part was inserted, deleted, or updated in the respective primary base tables.

A better solution might be to create a single-table join index on the columns of LineItem that need to be joined frequently with the Part table and then make the primary index for the join index l\_PartKey. Single-table join indexes are not cost-free, but the cost of performing the single row updates for a single-table index is far less expensive than the minijoins required by a multitable join index.

### Single-Table Join Index Definition

The definition for the join index might look something like this.

```
CREATE JOIN INDEX PartKeyLineItem AS
 SELECT l_PartKey, l_Quantity, l_SupplierKey
 FROM LineItem
 PRIMARY INDEX (l_PartKey);
```

The intent of defining this join index is to permit the Optimizer to select it in place of the base table LineItem in cases like the equality condition `l_PartKey = p_PartKey`, eliminating the need to redistribute the LineItem table (because its proxy, the join index table PartKeyLineItem, has the same primary index as that of the Part table, so the rows are stored on the same AMP). This avoids the large redistribution of LineItem, but not the join processing.

Not only can the Optimizer use single-table join indexes for rewriting queries, it can also use statistics collected on complex expressions in the index definition to better estimate single-table cardinalities. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information about using hash and single-table join indexes to estimate single-table cardinalities.

### General Procedure for Defining a Single-Table Join Index

1. Define a *column\_1\_name* for each *column\_name* in the primary base table to be included in the single-table join index.
2. To enhance join selectivity, define the primary index on a different column set than the primary base table, or define column partitioning.

3. If the physical database design warrants, use:
  - CREATE INDEX to create one or more NUSIs on the join index
  - A WHERE clause to define a sparse join index
  - A unique primary index for the join index

## Aggregate Join Indexes

An aggregate join index is a database object created using the CREATE JOIN INDEX statement, but specifying one or more columns that are derived from an aggregate expression. An aggregate join index is a join index that specifies MIN, MAX, SUM, COUNT, or that extracts a DATE value aggregate operations. No other aggregate functions are permitted in the definition of a join index; however, most of the other simple aggregate functions can be derived from these using column expressions. You can create aggregate join indexes as either single-table or as multitable join indexes. Aggregate join indexes can also be sparse (see [Sparse Join Indexes](#)).

### Functions of Aggregate Join Indexes

The primary function of an aggregate join index is to provide the Optimizer with a high-performing, cost-effective means for satisfying any query that specifies a frequently made aggregation operation on one or more columns.

In other words, aggregate join indexes permit you to define a persistent summary table without violating the normalization of the database schema. This allows a join index to precompute an aggregate value that would otherwise potentially require a table scan and sort operation.

Aggregate join indexes can be especially helpful for queries that roll up values for dimensions other than the primary key dimension, which would otherwise require redistribution.

An aggregate join index can be used to cover aggregate queries that only consider a subset of groups contained in the join index or have more join tables than the join index. In order to allow the aggregate join index to be used in this way, its definition must satisfy the following conditions:

- The grouping clause must include all columns that are specified in the grouping clause of the query.
- All columns in the query WHERE clause that join to tables not in the aggregate join index must be part of the join index definition.
- If you define row partitioning for an aggregate join index, its partitioning columns must be members of the column set specified in the GROUP BY clause of the index definition.

In other words, you cannot specify an *aggregated* column as a partitioning column.

- An aggregate join index cannot be column partitioned.

An aggregate join index can also be used to cover the following:

- Requests that specify COUNT(DISTINCT) and extended grouping such as CUBE, ROLLUP, and GROUPING SETS.
- Requests that specify subqueries or spooled derived tables.
- Both the outer and the inner tables in a request that specifies an outer join.

## Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

- You can create a global temporary table definition and then populate a materialized instance of it with aggregated result sets. This is not so much an alternate strategy as it is an entirely different strategy designed for an entirely different application.

Global temporary tables are private to the session that materializes them and their data does not persist beyond the end of that session. Unless you specify the `ON COMMIT PRESERVE` option when the definition for the temporary table is created, its contents do not persist even across individual database transactions. Unless provisions are made to write their contents to a persistent base table when a session ends, their data is not saved. Without numerous safeguards built into the process, this is not a method that provides any assurances about the integrity of the data it produces because while the contents of any global temporary table are private to the session in which it is created, the definition of the table is global within a database and any number of different sessions and users could materialize a different version of the table, populate it, and write the results to the same base table as any other session.

Note that the containing database or user for a global temporary table must have a minimum of 512 bytes of available PERM space to contain the table header for the GTT.

- You can create a volatile table with aggregate expressions defined on some or all its columns. The drawbacks of global temporary tables for this application apply equally to volatile tables.
- You can create a denormalized base table and populate it with an aggregated result set.

Denormalization always reduces the generality of the database for any ad hoc queries or data mining operations you might want to undertake as well as introducing various problematic update anomalies. While a relatively mild degree of denormalization is standard in physically implemented databases, the sort of denormalization called for by this solution is probably beyond what most DBAs would find acceptable, the enthusiasm of dimensional modeling theorists notwithstanding.

Because there is no mechanism for keeping such a table synchronized with its base table, it can become quickly outdated.

Nonetheless, for some applications this approach might be the only high-performing solution.

## Example

This example shows how a simple aggregate join index can be used to create prejoins with aggregation on one or more of its columns while retaining full normalization of the *physical* database. Strictly speaking, the term *normalization* applies only to the *logical* schema for a database, not to its physical schema; however, the term has unfortunately come to be used equally for both logical and physical database schemas. See [The Normalization Process](#) for details.

## Table Definitions

This example set uses the following table definitions.



```

CREATE TABLE customer (
 c_custkey INTEGER NOT NULL,
 c_name CHARACTER(26) CASESPECIFIC NOT NULL,
 c_address VARCHAR(41),
 c_nationkey INTEGER,
 c_phone CHARACTER(16),
 c_acctbal DECIMAL(13,2),
 c_mktsegment CHARACTER(21),
 c_comment VARCHAR(127))
UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders (
 o_orderkey INTEGER NOT NULL,
 o_custkey INTEGER,
 o_orderstatus CHARACTER(1) CASESPECIFIC,
 o_totalprice DECIMAL(13,2) NOT NULL,
 o_orderdate DATE FORMAT 'YYYY-MM-DD' NOT NULL,
 o_orderpriority CHARACTER(21),
 o_clerk CHARACTER(16),
 o_shippriority INTEGER,
 o_comment VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);

```

### Example Query Statement

Consider the following aggregate join query.

```

SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '2004-09-20'
AND o_orderdate < DATE '2004-10-15'
GROUP BY c_nationkey;

```

### Query Plan: No Aggregate Join Index Defined

Without an aggregate join index, a typical execution plan for this query might involve the following stages:

1. Redistribute orders into spool.
2. Sort the spool on o\_custkey.
3. Merge join the sorted spool and the customer file.
4. Aggregate the result of the merge join.

## Aggregate Join Index Definition

Suppose you define the following aggregate join index, which aggregates `o_totalprice` over a join of orders and customer.

```
CREATE JOIN INDEX ord_cust_idx AS
 SELECT c_nationkey, SUM(o_totalprice(FLOAT))
 AS price, o_orderdate
 FROM orders, customer
 WHERE o_custkey = c_custkey
 GROUP BY c_nationkey, o_orderdate
 ORDER BY o_orderdate;
```

## Query Plan for Aggregate Join Index

The execution plan produced by the Optimizer for this query includes an aggregate step on the aggregate join index, which is much smaller than either one of the join tables. You can confirm this by performing an EXPLAIN on the query.

```
EXPLAIN SELECT COUNT(*), SUM(o_totalprice)
 FROM orders, customer
 WHERE o_custkey = c_custkey
 AND o_orderdate > DATE '2005-09-20'
 AND o_orderdate < DATE '2005-10-15'
 GROUP BY c_nationkey;
```

```
*** Help information returned. 18 rows.
*** Total elapsed time was 3 seconds.
```

### Explanation

- ```
-----
```
- 1) First, we lock TPCD.ord_cust_idx for read on a reserved RowHash to prevent a global deadlock.
 - 2) Next, we do a SUM step to aggregate from join index table TPCD.ordcustidx by way of an all-rows scan with a condition of ("TPCD.ord_cust_idx.O_ORDERDATE > DATE '2005-09-20') AND (TPCD.ord_cust_idx.O_ORDERDATE < DATE '2005-10-15')", and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 2. The size of Spool 2 is estimated to be 1 row.
 - 3) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row. The estimated time for this step is 0.17 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Aggregate Join Index With EXTRACT Function

Join index definitions, both simple and aggregate, support the EXTRACT function. This example illustrates the use of the EXTRACT function in the definition of an aggregate join index.

Aggregate Join Index Definition

The index is defined as follows.

```
CREATE JOIN INDEX ord_cust_idx_2 AS
  SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price,
         EXTRACT(YEAR FROM o_orderdate) AS o_year
  FROM orders, customer
  WHERE o_custkey = c_custkey
  GROUP BY c_nationkey, o_year
  ORDER BY o_year;
```

The aggregation is based only on the year of o_orderdate, which has fewer groups than the entire o_orderdate, so ord_cust_idx_2 is much smaller than ord_cust_idx.

On the other hand, the use for ord_cust_idx_2 is more limited than ord_custidx. In particular, ord_cust_idx_2 can only be used to satisfy queries that select full years of orders.

Example Query Statement

While the join index defined for this example, ord_cust_idx_2, cannot be used for the query analyzed in *Query Plan for Aggregate Join Index* above, the following query does profit from its use because dates are on year boundaries.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND   o_orderdate > DATE '2004-01-01'
AND   o_orderdate < DATE '2004-12-31'
GROUP BY c_nationkey;
```

Sparse Join Indexes

Any join index, whether simple or aggregate, multitable or single-table, can be sparse. A sparse join index specifies a constant expression in the WHERE clause of its definition to narrowly filter its row population. For example, the following DDL creates an aggregate join index containing only the sales records from 2000:

```
CREATE JOIN INDEX j1 AS
  SELECT store_id, dept_id, SUM(sales_dollars) AS sum_sd
  FROM sales
  WHERE EXTRACT(year FROM sales_date) = 2000
  GROUP BY store_id, dept_id;
```

This method limits the rows included in the join index to a subset of the rows in the table based on an SQL request result.

When base tables are large, you can use this feature to reduce the content of the join index to only the portion of the table that is frequently used if the typical query only references a portion of the rows.

It is important to collect statistics on the sparse-defining column of the join index, or the Optimizer may not use the join index.

Sparse Join Indexes and Query Optimization

When a user query is entered, the Optimizer determines if accessing *j1* gives the correct answer and is more efficient than accessing the base tables. This sparse join index would be selected by the Optimizer only for queries that restricted themselves to data from the year 2000. For example, a query might require data from June of 2000. Because the join index *j1* contains all of the data for year 2000, it might be used to satisfy the following query:

```
SELECT store_id, dept_id, SUM(sales_dollars) AS sum_sd
FROM sales
WHERE sales_date >= 6/1/2000
AND    sales_date < 7/1/2000
GROUP BY storeid, dept_id;
```

As another example, the following DDL creates a join index containing only those customers living in four western states of the US:

```
CREATE JOIN INDEX westcust AS
SELECT cust.id, cust.address, donations.amount
FROM cust, donations
WHERE cust.cust_id = donations.cust_id
AND    donations.d_date > '2003/01/01'
AND    cust.state IN ('CA', 'OR', 'WA', 'NV');
```

The Optimizer can use this join index for the following query written to find all donors from California with donations of 1,000 USD or more made since January 1, 2003.

```
SELECT custid, cust.address
FROM cust, donations
WHERE donations.amount > 1000
AND    cust.state = 'CA'
AND    donations.d_date > '2003/01/01';
```

Because the customers and donations considered by the query are part of the subset included in the join index, the Optimizer uses it to answer the query. This can save a great deal of time especially in situations where the base tables are very large, but queries typically look only at subsets of the tables.

Maintenance of sparse join indexes can be much faster than maintenance for other join indexes because the sparse types have fewer values, and so are generally updated much less frequently.

See [Sparse Join Indexes and Tactical Queries](#) for information about design issues specific to sparse join index support for tactical queries.

Performance Impact of Sparse Join Indexes

A sparse index can focus on the portions of the tables that are most frequently used to do the following things.

- Make the costs for maintaining an index proportional to the percent of rows actually referenced in the index.
- Make request access faster where the resulting sparse index is smaller than a standard join index.
- Reduce the storage requirements for a join index where possible.

Using Outer Joins in Join Index Definitions

Join indexes should be defined using outer joins if they are intended to cover both inner and outer join queries. To understand how this is possible, an outer join can be thought of as producing a result consisting of two sets of rows. The first set corresponds to the set of matched rows obtained when a row from the outer table matches one or more rows from the inner table (this set corresponds to the set of rows defined by the inner join with the same join condition). The second corresponds to the set of unmatched rows: those rows from the outer table that do not match any rows from the inner table.

Except for the presence of the unmatched row set, an outer join is the same as an inner join, and produces the same result. Therefore if an inner join query can be completely satisfied by the matched set of rows from an outer join index, the Optimizer uses it.

Join Index With Outer Join in Its Definition

Consider the following join index defined on the three tables t1, t2, and t3. Tables t1 and t2 are joined with an inner join, and the result is joined with table t3 using an outer join. The outer table is the result of joining tables t1 and t2.

```
CREATE JOIN INDEX ji1 AS
SELECT a1, a2, a3
FROM (t1 INNER JOIN t2 ON a1 = a2)
LEFT OUTER JOIN t3 ON a1 = a3;
```

Column a3 is the unique primary index for table t3. Column a3 might be specified in the table definition explicitly as a primary index, or simply as unique. This means that all of the rows from the join of t1 with t2 are in the join index exactly once, either in the matched set, or in the unmatched set. Therefore, the following query can be satisfied by the join index:

```
SELECT a1, a2
FROM t1, t2
WHERE a1 = a2;
```

Extended Query Coverage With Outer Joins in Index Definition

A coverage algorithm determines that there is partial coverage, and the Optimizer uses the join index to join with the base tables to project the non-covered columns if the cost is lower than performing the query using the base tables alone.

The join index optimizations introduced by extending query coverage through defining extra foreign key-primary key joins does not negate the advantage of defining outer joins in your join index definitions because the normalization of outer joins to inner joins used by that optimization is specific to a particular class of queries. Queries that do not meet those specific criteria continue to benefit from the unnormalized outer join definitions in the join index.

Join indexes defined with outer joins can cover a query submitted in inner join format directly. Once the Optimizer converts the outer join in the query to an inner join by taking advantage of the equivalency of outer and inner joins for foreign key-primary key relationships (see [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)), the system can make a coverage test instead of identical matching. In other words, the predicates in the join index definition and in the query need not be identical.

More On Outer Join Index Coverage of Queries

A join index can be used to cover a wide variety of queries as long as the rows required in these queries form a subset of the row set contained in the join index. For example, consider a join index defined with the following SELECT query, where x1, x2 is a foreign key-primary key pair:

```
CREATE JOIN INDEX loj_cover AS
SELECT x1, x2
FROM t1, t2
WHERE x1=x2;
```

Any query of the following form can use this join index, where c represents any set of constant conditions:

```
SELECT x1, x2
FROM t1 LEFT OUTER JOIN t2 ON x1=x2 AND c;
```

This property greatly increases the applicability of many join indexes.

Both the join index and the query are normalized to inner joins when the original form is defined with an outer join and there is a foreign key-primary key relationship between the join column set (see [Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query](#)). The result is that a less restrictive coverage test can be applied to both the query and to the join index.

Using Outer Joins to Define Join Indexes

There are several benefits in defining non-aggregate join indexes with outer joins:

- Unmatched rows are preserved.
These rows allow the join index to satisfy queries with fewer join conditions than those used to generate the index.
- Outer table row scans can provide the same performance benefits as a single-table join index.
For example, the Optimizer can choose to scan the outer table rows of a join index to satisfy a query that only references the outer table provided that a join index scan would be more high-performing than scanning the base table or redistributing rows.

Redefined Join Index

The following example changes the previous join index example (see [Defining a Simple Join Index on a Binary Join Result](#)) to use an Outer Join in the join index definition.

```
CREATE JOIN INDEX OrdCustIdx AS
SELECT (o_custkey,c_name)
      ,
      (o_status,o_date,o_comment)
FROM orders LEFT JOIN customer ON o_custkey=c_custkey;
```

Materialized Join Index

The resulting join index rows would be the following (where the ? character indicates a null).

OrdCustIdx

Fixed Part		Repeated Part		
CustKey	Name	Status	Date	Comment
100	Robert	S	2004-10-01	big order
		S	2004-10-05	credit
101	Ann	P	2004-10-08	discount
102	Don	S	2004-10-01	rush order
		D	2004-10-03	delayed
?	?	U	2004-10-05	unknown customer

With the join index defined in this way, the following query could be resolved using just the join index, without having to scan the base tables.

```
SELECT o_status, o_date, o_comment
FROM orders;
```

In this particular case, it is more efficient to access the join index than it is to access the orders base table. This is true whenever the cost of scanning the join index is less than the cost of scanning the orders table. The Optimizer evaluates both access methods, choosing the more efficient, less costly of the two for its query plan.

Creating Join Indexes Using Outer Joins

This example examines the EXPLAIN reports for a different outer join-based join index in some detail.

Table Definitions

```
CREATE TABLE customer (
  c_custkey    INTEGER NOT NULL,
  c_name       CHARACTER(26),
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
UNIQUE PRIMARY INDEX(c_custkey);
CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus  CHARACTER(1),
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
  o_comment     VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
CREATE TABLE lineitem (
  l_orderkey    INTEGER NOT NULL,
  l_partkey     INTEGER NOT NULL,
  l_suppkey     INTEGER,
  l_linenumber  INTEGER,
  l_quantity    INTEGER NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount    DECIMAL(13,2),
  l_tax         DECIMAL(13,2),
```



```

l_returnflag    CHARACTER(1),
l_linestatus    CHARACTER(1),
l_shipdate      DATE FORMAT 'yyyy-mm-dd',
l_commitdate    DATE FORMAT 'yyyy-mm-dd',
l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
l_shipinstruct  VARCHAR(25),
l_shipmode      VARCHAR(10),
l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey);

```

Join Index Definition

The following statement defines a join index on these tables. Subsequent examples demonstrate the effect of this join index on how the Optimizer processes various queries.

```

CREATE JOIN INDEX order_join_line
  AS SELECT (l_orderkey, o_orderdate, o_custkey,
            o_totalprice),
            (l_partkey, l_quantity, l_extendedprice, l_shipdate)
  FROM lineitem LEFT JOIN orders ON l_orderkey = o_orderkey
  ORDER BY o_orderdate
  PRIMARY INDEX (l_orderkey);
*** Index has been created.
*** Total elapsed time was 15 seconds.

```

EXPLAIN for Query With Simple Predicate

The following EXPLAIN report shows how the newly created join index, order_join_line, might be used by the Optimizer.

```

EXPLAIN SELECT o_orderdate, o_custkey, l_partkey,
l_quantity,      l_extendedprice
  FROM lineitem, orders
 WHERE l_orderkey = o_orderkey;

```

Explanation

- 1) First, we lock LOUISB.order_join_line for read on a reserved Row Hash to prevent a global deadlock.
- 2) Next, we do an all-AMPs RETRIEVE step from join index table LOUISB.order_join_line by way of an all-rows scan with a condition of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 1, which is built locally on the AMPs. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 1 is estimated to be 1,000,000 rows. The estimated time for this step is 4 minutes and 27 seconds.
- 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

EXPLAIN for Query With More Complicated Predicate

The following EXPLAIN report shows how the join index might be used in a query when an additional search condition is added on the join indexed rows.

```
EXPLAIN SELECT o_orderdate, o_custkey, l_partkey,
              l_quantity, l_extendedprice
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '2003-11-01';
```

Explanation

- 1) First, we lock LOUISB.order_join_line for read on a reserved Row Hash to prevent a global deadlock.
- 2) Next, we do an all-AMPs RETRIEVE step from join index table LOUISB.order_join_line with a range constraint of ("LOUISB.order_join_line.Field_1026 > 971101") with a residual condition of ("(NOT (LOUISB.order_join_line.o_orderdate IS NULL)) AND (LOUISB.order_join_line.Field_1026 > 971101)") into Spool 1, which is built locally on the AMPs. The input table will not be cached in memory, but it is eligible for synchronized scanning. The size of Spool 1 is estimated to be 1000 rows. The estimated time for this step is 0.32 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

EXPLAIN for Query With Aggregation

The following EXPLAIN shows how the join index might be used in a query when aggregation is performed on the join indexed rows.

```
EXPLAIN SELECT l_partkey, AVG(l_quantity),
              AVG(l_extendedprice)
FROM lineitem , orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '2003-11-01'
GROUP BY l_partkey;
```

Explanation

- 1) First, we lock LOUISB.order_join_line for read on a reserved Row Hash to prevent a global deadlock.
- 2) Next, we do a SUM step to aggregate from join index table LOUISB.order_join_line with a range constraint of ("LOUISB.order_join_line.Field_1026 > 971101") with a residual condition of ("(LOUISB.order_join_line.Field_1026 > 971101) AND (NOT (LOUISB.order_join_line.o_orderdate IS NULL))"), and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The input table will not be cached in memory, but it is eligible for synchronized scanning.
- 3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 10 rows. The estimated time for this step is 0.32 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

EXPLAIN for Query With Base Table-Join Index Table Join

The following EXPLAIN shows how the join index might be used in a query when join indexed rows are used to join with another base table.

```
EXPLAIN SELECT o_orderdate, c_name, c_phone, l_partkey, l_quantity,
              l_extendedprice
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND   o_custkey = c_custkey;
```

Explanation

- 1) First, we lock LOUISB.order_join_line for read on a reserved Row Hash to prevent a global deadlock.
- 2) Next, we lock LOUISB.customer for read.
- 3) We do an all-AMPs RETRIEVE step from join index table LOUISB.order_join_line by way of an all-rows scan with a condition of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 2, which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated to be 1,000,000 rows. The estimated time for this step is 1 minute and 53 seconds.
- 4) We do an all-AMPs JOIN step from LOUISB.customer by way of a Row Hash match scan with no residual conditions, which is joined to Spool 2 (Last Use). LOUISB.customer and Spool 2 are joined using a merge join, with a join condition of ("Spool_2.o_custkey = LOUISB.customer.c_custkey"). The result goes into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 1,000,000 rows. The estimated time for this step is 32.14 seconds.
- 5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

EXPLAIN for Query Against Single Table

The following EXPLAIN report shows how the join index might be used in a query of a single table.

```
EXPLAIN SELECT l_orderkey, l_partkey, l_quantity, l_extendedprice
FROM lineitem
WHERE l_partkey = 1001;
```

Explanation

- 1) First, we lock LOUISB.order_join_line for read on a reserved Row Hash to prevent a global deadlock.
- 2) Next, we do an all-AMPs RETRIEVE step from join index table LOUISB.order_join_line by way of an all-rows scan with a condition of ("LOUISB.order_join_line.l_partkey = 1001") into Spool 1, which is built locally on the AMPs. The input table will not be cached in memory, but it is eligible for synchronized scanning. The result spool file will not be cached in memory. The size of Spool 1 is estimated to be 100 rows. The estimated time for this step is 59.60 seconds.
- 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

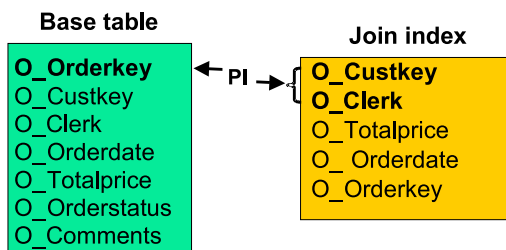
Join Indexes and Tactical Queries

Single-Table Join Indexes

One of the most useful constructs for tactical queries is the single-table join index. Because you can define a primary index for the join index composed of different columns than those used to define the base table primary index, you can create an alternative method of directly accessing data in the associated base table.

For example, you could create a join index on the orders table that includes only a subset of the columns that a particular tactical query application might require. In the example shown in the following graphic, assume that the application has available a value for o_custkey and the clerk that placed the order, but does not have a value for o_orderkey. Specifying the primary index defined for the join index OrderJI supports direct access to order base table using the single-table join index OrderJI.

```
CREATE JOIN INDEX OrderJI AS
  SELECT o_custkey, o_clerk, o_totalprice, o_orderdate, o_orderkey
  FROM orders
  PRIMARY INDEX(o_custkey,o_clerk);
```



A test against an orders table with 75 million rows, both with and without the OrderJI join index, returned the following response times from the query and join index above:

If the query is run ...	The response time is ...
without the join index	1:55.
with the join index	subsecond.

The larger the base table is, the longer it takes to process via a table scan, and the greater the benefit a join index providing single-AMP access provides. In this example, the join index took 8:23 to create.

A single-table join index like this one is particularly useful when the tactical application does not have the primary index of the base table available, but has an alternative row identifier. This might be the case when a social security number is available, but a member ID, the primary index of the base table, is not. Single-AMP access would still be achievable using the join index if its primary index is defined on social security number.

Aggregate Join Indexes

If your application supports repeated access to the same table using aggregation along the same set of dimensions, you should consider using aggregate join indexes to enhance the performance of those queries.

If you include one or more aggregating columns in the join index select list, then that index is an aggregate join index. Aggregate join indexes are dynamic summary tables, not snapshots. For example, the following aggregate join index computes a running sum on o_totalprice from the orders table:

```
CREATE JOIN INDEX ordersum AS
  SELECT o_clerk, SUM(o_totalprice) AS sumprice
  FROM orders
  GROUP BY o_clerk
  PRIMARY INDEX(o_clerk);
```

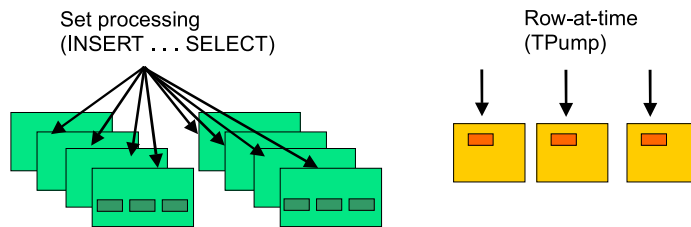
Each time the o_totalprice column of orders is updated, a new sum is computed and stored in the ordersum aggregate join index. If there are frequent queries throughout the day that request summaries of the total prices for orders placed by a specific clerk, this join index would be able to deliver response times suitable for tactical queries like the following example:

```
SELECT o_clerk, SUM(o_totalprice)
FROM orders
GROUP BY o_clerk
WHERE o_clerk = 'Clerk#000046240';
```

Use the GROUP BY column set as the primary index of the join index if this is the column whose value is specified by the application. Each query that specifies a value for that column will then be processed as a single-AMP request.

Join and Hash Index Maintenance Considerations

Each time a base table row is updated, its corresponding join or hash index data is modified within the same unit of work. Because MultiLoad and FastLoad are incompatible with join and hash indexes, maintenance of a base table that has a join or hash index must be performed using SQL, or the join or hash index must be dropped before the MultiLoad or FastLoad utility is run and the index then rebuilt afterward. As illustrated by the following figure, you can use SQL to take either a set processing approach or a row-at-a-time approach to join and hash index maintenance.



If an INSERT SELECT request is used to load data into the base table, then the unit of work is the entire SQL request. In such a case the join or hash index maintenance is performed in batch mode. Inserts to the base table are spooled and applied to the join or hash index structure in a single step of the query plan. In particular, when an INSERT SELECT request specifies the primary index value of the target row set in its select list or in its WHERE clause, a single-AMP merge step is used to process the INSERT operation.

When row-at-a-time updates are performed, the join or hash index structure is updated concurrently, once per base table row.

Join and hash index maintenance is optimized to use localized rowhash-level locking whenever possible. Table-level locks are applied when the maintenance is performed using all-AMPs operations such as spool merges.

When the table being updated contains a join or hash index, an EXPLAIN of the UPDATE request illustrates whether an all-AMPs operation and table-level locking are used at the time of the UPDATE operation, or a single-AMP operation with rowhash locking.

The following is a list of the conditions that support single-AMP updating and rowhash-level locking on join or hash indexes when the base table is updated a row-at-a-time. Be aware that these optimizations might not be applied to complicated indexes when cardinality estimates are made with low confidence or when the index is defined on three or more tables with all its join conditions based on nonunique and non-indexed columns.

When a row-at-a-time INSERT on the base table is being performed, the following restrictions apply:

- The join index can have a different primary index from the base table.
- Inserts that specify an equality constraint on the primary index column set of a table with joins between some non-primary index columns of the table and the primary index columns of another table are optimized to use rowhash-level locks. For example,

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
  SELECT x1,x2,x3,y2,y3,z2
  FROM t1,t2,t3
  WHERE x1=y1
  AND   y2=z1
  PRIMARY INDEX (y3);
```

When a row is inserted into t1 with the following INSERT statement,

```
INSERT INTO t1
VALUES (1,1,1);
```

corresponding j2 rows are materialized by a query like the following:

```
SELECT 1,1,1,y2,y3,z2
FROM t2,t3
WHERE y1=1
AND y2=z1;
```

If the number of rows that qualify `t2.y1=1` (see step 2-1) is within the 10% of the number of AMPs threshold and the number of rows resulting from the `t2` and `t3` join step 5 is also within this threshold, this INSERT statement does not incur any table-level locks.

For those INSERT operations that specify an equality constraint on a non-primary index or that involve joins with non-primary index columns, temporary hash indexes can be created by the system to process them in such a way that single-AMP retrievals are used and non-primary index joins are converted to primary index joins. See the following examples:

Given the following join index definition:

```
CREATE JOIN INDEX j1 AS
  SELECT x1,x2,x3,y2,y3
  FROM t1,t2
  WHERE x2= y1
  PRIMARY INDEX (x1);
```

When a row is inserted into t2 with the following INSERT statement

```
INSERT INTO t2
VALUES (1,1,1);
```

corresponding j1 rows are materialized by a query like the following:

```
SELECT x1, x2, x3 ,1 ,1
FROM t1
WHERE x2=1;
```

Given the following join index definition:

```
CREATE JOIN INDEX j1 AS
  SELECT x1,x2,x3,y2,y3
```

```
FROM t1,t2
WHERE x2=y1
PRIMARY INDEX (x1);
```

When a row is deleted from t2, delete t2 where t2.y1=1, corresponding j1 rows are materialized by a query like the following:

```
SELECT x1,x2,x3,y2,y3
FROM t1,t2
WHERE x2=y1
AND y1=1;
```

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
  SELECT x1,x2,x3,y1,y3
  FROM t1, t2
  WHERE x2=y2;
```

When a row is deleted from t2, delete t2 where t2.y1=1, corresponding j2 rows are materialized by a query which involves a non-PI-to-non-PI join:

```
SELECT x1,x2,x3,y1,y3
FROM t1, t2
WHERE x2=y2
AND y1=1;
```

If a single-table non-covering join index is defined on t1.x2, calling it stji_t1_x2, the (t1.x2=t2.y2) join that is processed by duplicating the qualified t2 row in step 4 to join with t1 in step 5 can be processed as follows:

1. a single-AMP retrieve from t2 by way of PI t2.y1=1 into Spool 2. Spool 2 is hash redistributed by t2.y2 and qualifies as a group-AMPs spool.
2. a few-AMPs join from Spool 2 to stji_t2_x2 on y2=x2, results going into Spool 3. Spool 3 is redistributed by t1.rowid and also qualifies as a group-AMPs spool.
3. a few-AMPs join back from Spool 3 to t1 on ROWID.

As long as the number of rows that qualify the join t1.x2=t2.y2 WHERE t1.y1=1 is within the 10% threshold, no table-level locks are incurred for the DELETE statement.

Given the following join index definition:

```
CREATE JOIN INDEX j3 AS
  SELECT x1,x2,x3,y2,y3,z2,z3
  FROM t1, t2, t3
  WHERE x2=y2
```



```

    AND    y1=z1
PRIMARY INDEX (y3);

```

When a row is inserted into t1, `INSERT INTO t1 VALUES (1,1,1)`, corresponding j3 rows are materialized by a query of the kind:

```

SELECT 1,1,1,y2,y3,z2,z3
FROM t2,t3
WHERE y2=1
AND    y1=z1

```

The index must be a single-table join index or hash index with the following exceptions:

- The join index primary index is composed of a column set from the target table of the simple insert.
- The join index primary index is composed of a column set that is joined, either directly or indirectly through transitive closure, to the target table of the simple insert.

In these cases, multitable join indexes are also optimized for single-AMP merge steps with rowhash-level locking.

For example, given the following tables, the join index definitions that follow qualify for single-AMP merge optimization:

```

CREATE TABLE t1 (
  x1 INTEGER,
  x2 INTEGER,
  x3 INTEGER)
PRIMARY INDEX (x1);

CREATE TABLE t2 (
  y1 INTEGER,
  y2 INTEGER,
  y3 INTEGER)
UNIQUE PRIMARY INDEX (y1);

CREATE TABLE t3 (
  z1 INTEGER,
  z2 INTEGER,
  z3 INTEGER)
PRIMARY INDEX (z1);

CREATE JOIN INDEX j1 AS
SELECT x1,x2,x3,y2,y3
FROM t1,t2
WHERE x2=y1

```

```

PRIMARY INDEX (x1);

CREATE JOIN INDEX j2 AS
  SELECT x1,x2,x3,y2,y3,z2,z3
  FROM t1,t2,t3
  WHERE x2=y1
  AND   y1=z1
PRIMARY INDEX (x2);

```

The following table indicates several base table inserts, their corresponding single-AMP merge join index inserts, and the qualifying condition that permits the single-AMP optimization:

Base Table Insert	Join Index Insert	Qualifying Condition
<pre> INSERT INTO t1 VALUES (1,1,1); </pre>	<pre> INSERT INTO j1 SELECT 1,1,1,y1,y3 FROM t2, t3 WHERE y1=1; INSERT INTO j2 SELECT 1,1,1,y1,y3,z1,z3 FROM t2,t3 WHERE y1=1 AND y1=z1; </pre>	The primary index of the join index is composed of a column set from the target table of the simple insert.
<pre> INSERT INTO t2 VALUES (1,1,1); </pre>	<pre> INSERT INTO j2 SELECT x1,x2,x3,1,1,z2,z3 FROM t1,t3 WHERE x2=1 AND 1=z1 AND x2=z1; </pre>	The primary index of the join index is composed of a column set that is joined directly or indirectly through transitive closure to the target table of the simple insert.
<pre> INSERT INTO t3 VALUES (1,1,1); </pre>	<pre> INSERT INTO j2 SELECT x1,x2,x3,y2,y3,1,1 FROM t1,t2 WHERE x2=y1 AND y1=1; </pre>	

- When a row-at-a-time UPDATE on the base table is being performed, the following restrictions apply:
 - The value for the primary index of the join index must be specified in the WHERE clause predicate of the request.
 - The UPDATE cannot change the primary index of the join index.
 - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using rowhash locks and a direct update step. If only a few rows are updated (a few-AMPs operation), rowhash READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash locks are also applied to the base table using the rowID values extracted from the index rows.

- When a row-at-a-time DELETE on the base table is performed, the following restrictions apply:
 - The value for the primary index of the join index must be specified in the WHERE clause predicate of the DELETE statement.
 - The deleted row must not be from the inner table of an outer join in the CREATE JOIN INDEX statement with the following exceptions:
 - The outer join condition in the join index is specified on a UPI column from the inner table.
 - The outer join condition in the join index is specified on a NUPI column from the inner table.
 - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using rowhash-level locks and a direct delete step. If only a few rows are deleted (a few-AMPs operation), rowhash-level READ locks are placed on the NUSI subtable for the index rows that are read. Rowhash-level locks are also applied to the base table using the rowID values extracted from the index rows.

Under all other conditions, a single-row update causes a table-level WRITE lock to be placed on the hash or join index.

If table-level locks are reported in the EXPLAIN text, then consider using set processing approaches with one or more secondary indexes as an alternative.

Examples of Row-at-a-Time INSERT Maintenance Overhead

Using the guidelines provided in *Join and Hash Index Maintenance Considerations* above, you can create a single-table join index that performs well with continuous row-at-a-time inserts to its base table. The maintenance to that join index structure, if Teradata Parallel Data Pump were performing continuous inserts, would be one additional single-AMP operation accompanied by a single rowhash-level WRITE lock.

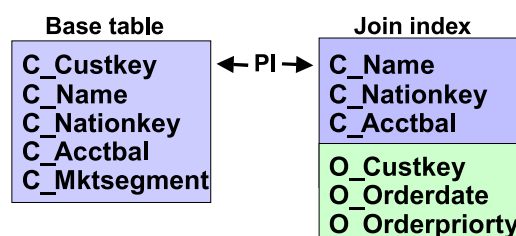
A further example of join index maintenance shows that row-at-a-time inserts into a base table that participates in a multitable join index exert a table-level lock or a partition-level lock (if partition elimination is possible) on the join index except for the following cases:

- The primary index of the join index is composed of a column set from the target table of the simple insert.
- The primary index of the join index is composed of a column set that is joined either directly or indirectly through transitive closure to the target table of the simple insert.

In the following example, the primary index of a row-compressed multitable join index is a subset of the base table being updated, so a single-AMP merge can be used.

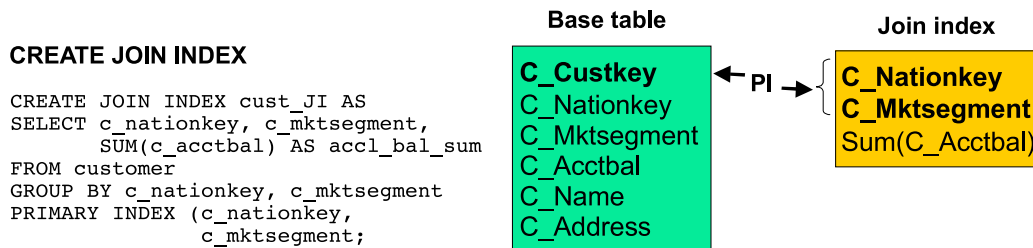
CREATE JOIN INDEX

```
CREATE JOIN INDEX JImulti AS
SELECT (o_custkey, c_nationkey,
       c_acctbal, c_name),
       (a_orderdate, o_orderpriority)
FROM orders, customer
WHERE o_custkey = c_custkey
PRIMARY INDEX (c_name);
```



If neither exception is true, then consider using set processing approaches to update the base table (see *Set Processing Alternative* below).

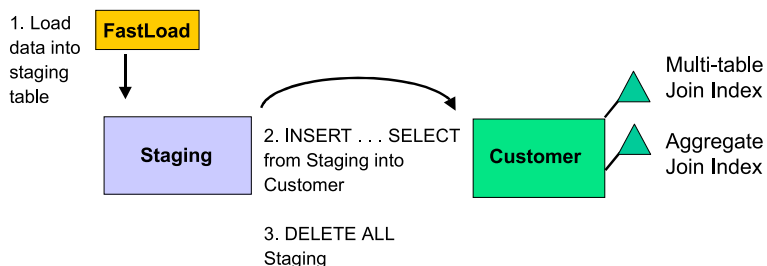
An aggregate join index incurs the same maintenance overhead as a multitable join index for row-at-a-time base table inserts. The following example uses a single-table aggregate join index.



For multitable join indexes, the Optimizer uses rowhash-level locks wherever possible, resulting in fewer table-level locks on the join index table. If there are just a few rows in the join index that are impacted, the Optimizer places several rowhash-level locks on just those AMPs and then uses a group-AMPs, rather than an all-AMPs operation.

Set Processing Alternative

For situations where row-at-a-time maintenance imposes table-level locking, consider updating the base table with periodic set processing approaches. The following example shows a set processing approach to updating the customer table, using frequent, short INSERT SELECT operations into the base table that also update its associated multitable and aggregate join indexes.



Join Index Definition Restrictions

This topic lists several restrictions on join index definitions. For complete details about join index syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Restrictions on Number of Join Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table, in any combination, is 32. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints. Each composite NUSI that specifies an ORDER BY clause counts as 2 consecutive indexes in this calculation (see [Importance of Consecutive Indexes for Value-Ordered NUSIs](#)). You cannot define join, or any other, indexes on global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184).

Suppose you have four tables, each with multiple secondary, hash, and join indexes defined on them:

- table_1 has 32 secondary indexes and no hash or join indexes.
- table_2 has 16 secondary indexes, no hash indexes, and 16 join indexes.
- table_3 has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- table_4 has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but they all operate at the boundaries of the defined limits.

Note that if any of the secondary indexes defined on tables 1, 2, or 3 is a composite NUSI defined with an ORDER BY clause, the defined limits are exceeded, and the last index you attempt to create on the table will fail. Because each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in the count against the maximum of 32 per table, you could define only 8 of them on table_2, for example, if you also defined 16 join indexes on the table.

Restrictions on the Number of Columns Per Referenced Base Table

The following restrictions apply to the number of columns per table that can be specified in a join index definition.

- The maximum number of columns that can be specified in a join index per referenced base table is 64.
- The maximum number of columns that can be specified in a multitable join index definition is 2,048.
- When you specify join index row compression, then each of the *column_1* and *column_2* sets is limited to 64 common base table references.

Restrictions on the Use of the System-Derived PARTITION[#L n] Column

You cannot use a system-derived PARTITION[#L n] column in a join index definition.

Restrictions on Outer Join Definitions

The following restrictions apply to outer joins when used to define a join index.

- FULL OUTER JOIN is not valid.
- When you specify a LEFT or RIGHT outer join, the following rules apply:
 - The outer table joining column for each condition must be contained in either *column_1_name* or *column_2_name*.
 - The inner table of each join condition must have at least one non-nullable column in either *column_1_name* or *column_2_name*.

Restrictions on Secondary Index Definitions

You cannot define unique secondary indexes on a join index.

You can define the primary index of a join index to be row partitioned if and only if the join index is not also defined with row compression. For more information about specifying partitioned primary indexes for join indexes, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Restrictions on Built-In Functions and Join Index Definitions

When you create a join index that specifies a built-in, or system, function in its WHERE clause (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for more information about built-in functions), the system resolves the function at the time the join index is created and then stores the result as part of the index definition rather than evaluating it dynamically at the time the Optimizer would use the index to build a query plan.

As a result, the Optimizer does not use a join index in the access plan for a query that qualifies its WHERE clause with the same built-in function used to define that join index because it cannot determine whether the index covers the query or not. The Optimizer does use the join index if the query specifies an explicit value in its WHERE clause that matches the resolved value stored in the index definition.

(The exception to this is the CURRENT_TIME and CURRENT_TIMESTAMP functions, which are resolved for a query. The resolved value is used to check if the query can be covered by a join index.)

For example, suppose you decide to define a join index using the CURRENT_DATE built-in function on January 4, 2010 as follows:

```
CREATE JOIN INDEX curr_date AS
  SELECT *
  FROM orders
  WHERE order_date = CURRENT_DATE;
```

On January 7, 2010, you perform the following SELECT statement:

```
SELECT *
FROM orders
WHERE order_date = CURRENT_DATE;
```

When you EXPLAIN this query, you find that the Optimizer does not use join index curr_date because the date stored in the index definition is the explicit value '2010-01-04', not the current system date '2010-01-07'.

Note that if you had defined curr_date with a predicate of order_date >= CURRENT_DATE instead of order_date = CURRENT_DATE, then the Optimizer could use curr_date to cover the query if it were the least costly way to process the request.

On the other hand, if you were to perform the following SELECT statement on January 7, 2010, or any other date, retaining the original curr_date predicate, the Optimizer does use join index curr_date for the query plan because the statement explicitly specifies the same date that was stored with the join index definition when it was created:

```
SELECT *
  FROM orders
  WHERE order_date = DATE '2010-01-04';
```

Restriction on Number of Join Indexes Selected Per Query

The Optimizer can use several join indexes for a single query, selecting one multitable join index as well as additional single-table join indexes for its join plan. The join indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable join indexes for the plan. Always examine your EXPLAIN reports to determine which join indexes are used for the join plans generated for your queries. If a join index you think should have been used by a query was not included in the join plan, try restructuring the query and then EXPLAIN it once again.

The limit on the number of join indexes considered per query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort. In other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

Restrictions on Partial Covering by Join Indexes

The Optimizer can use a join index that partially covers a query in the following cases:

- One of the columns in the index definition is the keyword ROWID.
You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).
- The column set defining the UPI of the underlying base table is also carried in the definition.
- The column set defining the NUPI of the underlying base table plus either of the following is also carried in the definition:
 - One of the columns in the definition of that index is the keyword ROWID.
You can only specify ROWID in the outermost SELECT of a CREATE JOIN INDEX request.
 - The column set defining a USI on the underlying base table.
The ROWID option is the preferable choice.
- Partial covering is not supported for the inner table of an outer join.
- Partial covering is not supported for queries that contain a TOP *n* or TOP *m* PERCENT clause.

If statistics indicate that it would be cost-effective, the Optimizer can specify that the partially covering single-table join index be joined to one of its underlying base tables using either the ROWID or the UPI or USI to join to the column data not defined for the index itself.

Even though you do not explicitly specify this join when you write your query, it counts against the 128 table restriction on joins.

For example, suppose you define the tables t1, t2, and t3 and the join indexes j1 and j2 as follows:

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER);
```

```

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER);

CREATE TABLE t3 (
  a3 INTEGER,
  b3 INTEGER,
  c3 INTEGER);

CREATE JOIN INDEX j1 AS
  SELECT b1, b2, t1.ROWID AS t1rowid
  FROM t1,t2
  WHERE a1=a2;

CREATE JOIN INDEX j2 AS
  SELECT b1, b2, t1.ROWID t1rowid, t2.rowid t2rowid
  FROM t1,t2
  WHERE a1=b2;

```

Join index j1 partially covers the following queries:

```

SELECT a1, b1, c1, b2
FROM t1,t2
WHERE t1.a1=t2.b2
AND   t1.b1=10;

SELECT a1, b1, c1, b2, b3, c3
FROM t1,t2,t3
WHERE t1.a1=t2.b2
AND   t1.b1=t3.a3
AND   t3.b3 > 0;

```

The same join index does not partially cover the following queries:

```

SELECT *
FROM t1,t2
WHERE t1.a1=t2.b2
AND   t1.b1=10;

SELECT *
FROM t1,t2,t3
WHERE t1.a1=t2.b2

```



```

AND    t1.b1=t3.a3
AND    t3.b3 > 0;

```

Join index j2, on the other hand, partially covers all of these queries.

Even though you do not explicitly specify the join back to the base table when you write your query, it counts against the 64 tables per query block restriction on joins.

Be aware that a join index defined with an expression in its select list provides less coverage than a join index that is defined using base columns.

For example, the Optimizer can use join index ji_f1 to rewrite a query that specifies any character function on f1.

```

CREATE JOIN INDEX ji_f1 AS
  SELECT b1, f1
  FROM t1
  WHERE a1 > 0;

```

At the same time, because it is defined with a SUBSTR expression in its select list, the Optimizer can only use join index ji_substr_f1 to rewrite a query that specifies the same SUBSTR function on f1.

```

CREATE JOIN INDEX ji_substr_f1 AS
  SELECT b1, SUBSTR(f1,1,10) AS s
  FROM t1
  WHERE a1 > 0;

```

Restriction on Coverage by Join Indexes When a Join Index Definition References More Tables Than a Query

Whether the Optimizer decides to include a join index in its query plan is a more complicated choice than simply determining if the index contains all the table columns specified in the query. The columns on which the base tables in the index definition are joined and their respective referential constraints also play an important role. See [Rules for Whether Join Indexes With Extra Tables Cover Queries](#).

In many cases, the Optimizer does not consider using a join index in its access plan if that index is defined on more tables than the query references. This is because the so-called extra inner joins involving the tables not referenced by the query can cause both spurious row loss and spurious duplicate row creation during the optimization process, and either outcome produces incorrect results.

This outcome can be avoided, and the join index used in the query access plan, if the extra inner joins are defined on primary key-foreign key relationships in the underlying base tables that ensure proper row preservation. Such a join index is referred to as a broad join index. The referential integrity relationship between the base table primary and foreign keys can be specified using any of the three available methods for establishing referential constraints between tables. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

A broad join index is a covering join index whose definition includes one or more tables that is not specified in the query it covers. A wide range of queries can make use of a broad join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the index to be used to cover queries over a subset of dimension tables.

In the case of join index outer joins, outer table rows are always preserved automatically, so there is no requirement for a referential integrity constraint to exist in order to preserve them.

In the case of foreign key-primary key inner joins, the same preservation of rows follows from a declarative referential integrity constraint. In this case, the Optimizer does consider a join index with extra inner joins in its definition to cover a query. The following paragraphs explain why a referential constraint preserves logical integrity:

Assume that the base tables in a join index definition can be divided into two distinct sets, s1 and s2.

s1 contains the base tables referenced in the query, while s2 contains the extra base tables the query does not reference. The base tables in s1 are joined to the base tables in s2 on foreign key-primary key columns, with the tables in s2 being the primary keys in the relationships. The foreign key values cannot be null because if the foreign key column set contains nulls, the losslessness of the foreign key table cannot be guaranteed.

The following assertions about these base tables are true:

- The extra joins do not eliminate valid rows from the join result among the base tables in s1 because FOREIGN KEY and NOT NULL constraints ensure that every row in the foreign key table finds its match in the primary key table.
- The extra joins do not introduce duplicate rows in the join result among the base tables in s1 because the primary key is, by definition, unique and not nullable.

These assertions are also true for extra joins made between base tables that are both in s2.

Therefore, the extra joins in a join index definition, if made on base tables that are defined in a way that observes these assertions, preserve all the rows resulting from the joins among the base tables in s1 and do not add spurious rows. This result permits the Optimizer to use the join index to cover a query that references fewer tables than the index definition inner joins together.

The Optimizer can select a join index for a query plan if the index contains either the same set, or a subset, of the tables referenced by the query. If more tables are referenced in the join index definition than are referenced by the query, the Optimizer generally does not consider that index as a candidate for coverage because the extra joins can either eliminate rows or produce duplicate rows or both. Because a referential integrity relationship guarantees the losslessness of the foreign key table in the join with its primary key table, extra tables in a join index definition do not disqualify it from query plan consideration if the extra joins allow the join index to preserve all the rows for the join result of the subset of tables in the query.

For example, suppose you define a join index on a set of 5 tables, t1 - t5, respectively, with foreign key-primary key joins in the directions indicated (arrows point from a foreign key table to its parent primary key table) by the following diagram:

t1 → t2 → t3 → t4 → t5

Queries that reference the following table subsets can be covered by this join index because the extra joins, either between two tables where one is in the query and the other is not, or between two tables that are both not in the query, do not cause any loss of rows for the join result of the subset of tables in the query:

- t1
- t1, t2
- t1, t2, t3
- t1, t2, t3, t4

As a result of this property, the following conditions that reference extra joins can be exploited by the Optimizer when the number of tables referenced by the join index definition exceeds the number of tables referenced by the query. In each case, x1 is a unique RI-related column in table t1 and x2 is a unique RI-related column in table t2:

Join Index Extra Join Condition	Qualifications
x1 = x2	<ul style="list-style-type: none"> • x1 is the foreign key in the relationship. • t1 is referenced by the query. • t2 is not referenced by the query.
x1 = x2	<ul style="list-style-type: none"> • x1 is the primary key in the relationship. • t2 is referenced by the query. • t1 is not referenced by the query.
x1 = x2	<ul style="list-style-type: none"> • x1 is the foreign key in the relationship. • x2 is the primary key in the relationship. • Neither t1 nor t2 is referenced by the query.
x1 = x2	<ul style="list-style-type: none"> • x1 is the primary key in the relationship. • x2 is the foreign key in the relationship. • Neither t1 nor t2 is referenced by the query.

One restriction with two critical exceptions must be added to the above optimization to make the coverage safe: when one table referenced in the join index definition is the parent table of more than one FK table, the join index is generally disqualified from covering any query that references fewer tables than are referenced in the join index.

For example, suppose you define a join index on a set of 5 tables, t1 - t5, respectively, with foreign key-primary key joins in the directions indicated (arrows point from a foreign key table to its parent primary key table) by the following diagram:

t1 → t2 → t3 → t4 ← t5

For these RI relationships, table t4 is the parent table of both tables t3 and t5. The losslessness of the foreign key table depends on the fact that the parent table has all the primary keys available. Joining the parent table with another child table can render this premise false. Therefore, the final join result cannot be viewed as lossless for any arbitrary subset of tables.

The following two points are exceptions to this restriction:

- When the foreign key tables are joined on the foreign key columns, there is no loss on any of the foreign key tables because they all reference the same primary key values in their common parent table.
- All foreign key tables reference the same primary key column in the primary key table. By transitive closure, all these foreign key tables are related by equijoins on the foreign key columns.

By specifying extra joins in the join index definition, you can greatly enhance its flexibility.

For example, suppose you have a star schema based on a Sales fact table and the following dimension tables:

- Customer
- Product
- Location
- Time

You decide it is desirable to define a join index that joins the fact table Sales to its various dimension tables in order to avoid the relatively expensive join processing between the fact table and its dimension tables whenever ad hoc join queries are made against them.

If there are foreign key-primary key relationships between the join columns, which is often the case, the join index can also be used to optimize queries that only reference a subset of the dimension tables.

Without taking advantage of this optimization, you must either create a different join index for each category of query, incurring the greater cost of maintaining multiple join indexes, or you lose the benefit of join indexes for optimizing the join queries on these tables altogether. By exploiting the foreign key-primary key join properties, the same join index can be selected by the Optimizer to generate access plans for a wide variety of queries.

As is always true, even if this optimization can be used for a given situation, the plan produced using it is compared with other generated plans, and the least costly plan from the generated plan set is the one that the system uses.

Rules for Whether Join Indexes With Extra Tables Cover Queries

The following rules explain how to design a set of underlying base tables for a join index definition in such a way to ensure that the Optimizer selects the index for an access plan if it inner joins more tables than the query references.

IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ...	THEN the Optimizer ...
the extra joins are not made on foreign key-primary key columns in the underlying base tables	does not consider the join index for the query plan. This is because the presence of extra joins in the definition can either eliminate existing rows from the query evaluation or produce duplicate rows during optimization.
the extra joins are made on foreign key-primary key columns in the underlying base tables	considers the join index for use in for the query plan.

IF there are more inner-joined tables in a join index definition than the number of tables referenced in a query and ...

THEN the Optimizer ...

- both of the following conditions are true:
- The join column set of the inner table in the extra outer join is unique
 - Either the inner table or both the inner and outer tables involved in the extra outer join are extra tables

Examples That Obey the General Covering Rules for Extra Tables in the Join Index Definition

The following set of base tables, join indexes, queries, and EXPLAIN reports demonstrate how the referential integrity relationships among the underlying base tables in a join index definition influence whether the Optimizer selects the index for queries that reference fewer base tables than are referenced by the join index.

```
CREATE SET TABLE t1, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL (
  x1 INTEGER NOT NULL,
  a1 INTEGER NOT NULL,
  b1 INTEGER NOT NULL,
  c1 INTEGER NOT NULL,
  d1 INTEGER NOT NULL,
  e1 INTEGER NOT NULL,
  f1 INTEGER NOT NULL,
  g1 INTEGER NOT NULL,
  h1 INTEGER NOT NULL,
  i1 INTEGER NOT NULL,
  j1 INTEGER NOT NULL,
  k1 INTEGER NOT NULL,
  CONSTRAINT ri1 FOREIGN KEY (a1, b1, c1) REFERENCES t2 (a2,b2,c2),
  CONSTRAINT ri2 FOREIGN KEY (d1) REFERENCES t3(d3),
  CONSTRAINT ri3 FOREIGN KEY (e1,f1) REFERENCES t4 (e4,f4),
  CONSTRAINT ri4 FOREIGN KEY (g1,h1,i1,j1) REFERENCES t5(g5,h5,i5,j5),
  CONSTRAINT ri5 FOREIGN KEY (k1) REFERENCES t6(k6));

CREATE SET TABLE t2, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL (
  a2 INTEGER NOT NULL,
  b2 INTEGER NOT NULL,
  c2 INTEGER NOT NULL,
  x2 INTEGER)
UNIQUE PRIMARY INDEX(a2, b2, c2);

CREATE SET TABLE t3, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL (
  d3 INTEGER NOT NULL,
  x3 INTEGER)
UNIQUE PRIMARY INDEX(d3);

CREATE SET TABLE t4, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL (
  e4 INTEGER NOT NULL,
  f4 INTEGER NOT NULL,
  x4 INTEGER)
UNIQUE PRIMARY INDEX(e4, f4);

CREATE SET TABLE t5, NO BEFORE JOURNAL,
```

```

                                NO AFTER JOURNAL (
g5 INTEGER NOT NULL,
h5 INTEGER NOT NULL,
i5 INTEGER NOT NULL,
j5 INTEGER NOT NULL,
x5 INTEGER)
UNIQUE PRIMARY INDEX(g5, h5, i5, j5);

CREATE SET TABLE t6, NO BEFORE JOURNAL,
                                NO AFTER JOURNAL (
    k6 INTEGER not null,
    x6 INTEGER)
UNIQUE PRIMARY INDEX(k6);

```

Example: All Outer Joins in Join Index Definition

The following join index definition left outer joins t1 to t3 on da=d3 and then left outer joins that result to t6 on k1=k6.

```

CREATE JOIN INDEX jiout AS
  SELECT d1, d3, k1, k6, x1
  FROM t1 LEFT OUTER JOIN t3 ON d1=d3
        LEFT OUTER JOIN t6 ON k1=k6;

```

You would expect the Optimizer to use jiout with the following query, because all the outer joins in the join index are inner joined to the query tables on unique columns (d1 and k1 are declared foreign keys in t1 and d3 and k6, the primary keys for t3 and t6, respectively, are declared as the unique primary index for those tables).

The bold EXPLAIN report text indicates that the Optimizer does use jiout in its query plan.

```

EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
  WHERE d1=d3
  GROUP BY 1;
*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.

```

Explanation

- 1) First, we lock **HONG_JI.jiout** for read on a reserved RowHash to prevent a global deadlock.
- 2) Next, we do an all-AMPs SUM step to aggregate from **HONG_JI.jiout** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
- 3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1 row. The estimated time for this step is 0.04 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
 - > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: All Inner Joins Without Aggregation in Join Index Definition

The following simple join index definition specifies inner joins on tables t1, t3 and t6.

```
CREATE JOIN INDEX jiin AS
  SELECT d1, d3, k1, k6, x1
  FROM t1, t3, t6
  WHERE d1=d3
  AND    k1=k6;
```

You would expect the Optimizer to use jiin with the following query, and the bold EXPLAIN report text indicates that it does.

```
EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
  WHERE d1=d3
  GROUP BY 1;
```

```
*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

Explanation

- 1) First, we lock **HONG_JI.jiin** for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we do an all-AMPS SUM step to aggregate from **HONG_JI.jiin** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
 - 3) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with low confidence to be 1 row. The estimated time for this step is 0.04 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: All Inner Joins With Aggregation in Join Index Definition

The following aggregate join index definition specifies inner joins on tables t1, t2, and t4.

```
CREATE JOIN INDEX jiin_aggr AS
  SELECT a1, e1, SUM(x1) AS total
  FROM t1, t2, t4
  WHERE a1=a2
  AND    e1=e4
  AND    a1>1
  GROUP BY 1, 2;
```

You would expect the Optimizer to use join index `jiin_aggr` in its plan for the following query because it has the same join term as the query.

The bold EXPLAIN report text indicates that the Optimizer does use `jiin_aggr` in its plan:

```
EXPLAIN SELECT a1, e1, SUM(x1) AS total
  FROM t1, t2, t4
 WHERE a1=a2
    AND e1=e4
    AND a1>2
 GROUP BY 1, 2;

*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.
Explanation
-----
1) First, we lock HONG_JI.jiin_aggr for read on a reserved RowHash
to prevent a global deadlock.
2) Next, we do an all-AMPS RETRIEVE step from HONG_JI.jiin_aggr by way of
an all-rows scan with a condition of ("HONG_JI.jiin_aggr.a1 > 2)
AND (HONG_JI.jiin_aggr.a1 >= 3)") into Spool 1 (group_amps), which
is built locally on the AMPS. The size of Spool 1 is estimated
with no confidence to be 1 row. The estimated time for this step
is 0.03 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPS involved
in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
statement 1. The total estimated time is 0.03 seconds.
```

Example: All Inner Joins With Aggregation in Join Index Definition

You would not expect the Optimizer to use join index `jiin_aggr` (see *Example: All Inner Joins With Aggregation in Join Index Definition* above) in its plan for the following query because the condition `b1=b2` AND `f1=f4` is not covered by the join index defined by `jiin_aggr`. As a result, the Optimizer specifies a full-table scan to retrieve the specified rows.

The EXPLAIN report text indicates that the Optimizer does not choose `jiin_aggr` to cover the query:

```
EXPLAIN SELECT a1, e1, SUM(x1) AS total
  FROM t1, t2, t4
 WHERE b1=b2
    AND f1=f4
    AND a1>2
 GROUP BY 1, 2;

Explanation
-----
1) First, we lock HONG_JI.t4 for read on a reserved RowHash to
prevent a global deadlock.
2) Next, we lock HONG_JI.t2 for read on a reserved RowHash to
prevent a global deadlock.
3) We lock HONG_JI.t1 for read on a reserved RowHash to prevent a
global deadlock.
4) We do an all-AMPS RETRIEVE step from HONG_JI.t1 by way of an
all-rows scan with a condition of ("HONG_JI.t1.a1 > 2") into Spool
4 (all_amps), which is duplicated on all AMPS. The size of Spool
4 is estimated with no confidence to be 2 rows. The estimated
time for this step is 0.03 seconds.
5) We do an all-AMPS JOIN step from HONG_JI.t2 by way of an all-rows
scan with no residual conditions, which is joined to Spool 4 (Last
Use). HONG_JI.t2 and Spool 4 are joined using a product join,
```


- with a join condition of ("b1 = HONG_JI.t2.b2"). The result goes into Spool 5 (all_amps), which is duplicated on all AMPs. The size of Spool 5 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.04 seconds.
- 6) We do an all-AMPs JOIN step from HONG_JI.t4 by way of an all-rows scan with no residual conditions, which is joined to Spool 5 (Last Use). HONG_JI.t4 and Spool 5 are joined using a product join, with a join condition of ("f1 = HONG_JI.t4.f4"). The result goes into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
 - 7) We do an all-AMPs SUM step to aggregate from Spool 3 (Last Use) by way of an all-rows scan, and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 6. The size of Spool 6 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.05 seconds.
 - 8) We do an all-AMPs RETRIEVE step from Spool 6 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
 - 9) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
 - > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: More Inner Joined Tables in Aggregate Join Index Definition Than in Query

The following aggregate join index definition specifies inner joins on tables t1, t3, and t6 using conditions that exploit a foreign key-primary key relationship between table t1 and tables t3 and t6, respectively.

```
CREATE JOIN INDEX jiin_aggr AS
  SELECT d1, k1, SUM(x1) AS total
  FROM t1, t3, t6
  WHERE d1=d3
  AND    k1=k6
  GROUP BY 1, 2;
```

You would expect the Optimizer to include join index jiin_aggr in its access plan for the following query even though jiin_aggr is defined with an inner joined table, t6, that the query does not reference. This is acceptable to the Optimizer because of the foreign key-primary key relationship between t1 and the extra table, t6, on columns k1 and k6, which have a foreign key-primary key relationship and are explicitly defined as a foreign key and as the unique primary index for their respective tables.

The bold EXPLAIN report text indicates that the Optimizer does select jiin_aggr for the query plan:

```
EXPLAIN SELECT d1, SUM(x1)
  FROM t1, t3
  WHERE d1=d3
  GROUP BY 1;
```

```
*** Help information returned. 17 rows.
*** Total elapsed time was 1 second.
```

Explanation

- 1) First, we lock **HONG_JI.jiin_aggr** for read on a reserved RowHash to prevent a global deadlock.

- 2) Next, we do an all-AMPs SUM step to aggregate from **HONG_JI.jiin_aggr** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row. The estimated time for this step is 0.03 seconds.
 - 3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1 row. The estimated time for this step is 0.04 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: Join Index Left Outer Joined on Six Tables

The following join index definition left outer joins table t1 with, in succession, tables t2, t3, t4, t5, and t6 on a series of equality conditions made on foreign key-primary key relationships among the underlying base tables.

```
CREATE JOIN INDEX jiout AS
  SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, h1, i1, j1, j5,
         k1, k6, x1
  FROM t1
  LEFT OUTER JOIN t2 ON a1=a2 AND b1=b2 AND c1=c2
  LEFT OUTER JOIN t3 ON d1=d3
  LEFT OUTER JOIN t4 ON e1=e4 AND f1=f4
  LEFT OUTER JOIN t5 ON g1=g5 AND h1=h5 AND i1=i5 AND j1=j5
  LEFT OUTER JOIN t6 ON k1=k6;
```

Even though the following query references fewer tables than are defined in the join index, you would expect the Optimizer to include join index **ji_out** in its access plan because all the extra outer joins are defined on unique columns and the extra tables are the inner tables in the outer joins.

The bold EXPLAIN report text indicates that the Optimizer does select **ji_out** for the query plan:

```
EXPLAIN SELECT a1, b1, c1, SUM(x1)
  FROM t1, t2
  WHERE a1=a2
  AND   b1=b2
  AND   c1=c2
  GROUP BY 1, 2, 3;
```

```
*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

Explanation

- 1) First, we lock **HONG_JI.jiout** for read on a reserved RowHash to prevent a global deadlock.
- 2) Next, we do an all-AMPs SUM step to aggregate from **HONG_JI.jiout** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 2 rows. The estimated time

- for this step is 0.03 seconds.
- 3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: Many More Tables Referenced by Join Index Definition Than Referenced by Query

The following join index definition specifies all inner joins on tables t1, t2, t3, t4, t5 and t6 and specifies equality conditions on all the foreign key-primary key relationships among those tables.

```
CREATE JOIN INDEX ji_in AS
  SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, g5, h1, i1, j1,
         k1, k6, x1
  FROM t1, t2, t3, t4, t5, t6
 WHERE a1=a2
  AND  b1=b2
  AND  c1=c2
  AND  d1=d3
  AND  e1=e4
  AND  f1=f4
  AND  g1=g5
  AND  h1=h5
  AND  i1=i5
  AND  j1=j5
  AND  k1=k6;
```

Even though 6 tables are referenced in the join index definition, and all its join conditions are inner joins, you would expect the Optimizer to include join index ji_in in its query plan for the following query, which only references 2 of the 6 tables, because all the conditions in the join index definition are based on foreign key-primary key relationships among the underlying base tables.

The bold EXPLAIN report text indicates that the Optimizer does select ji_in for the query plan:

```
EXPLAIN SELECT a1, b1, c1, SUM(x1)
  FROM t1, t2
  WHERE a1=a2
  AND  b1=b2
  AND  c1=c2
  GROUP BY 1, 2, 3;
*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

Explanation

-
- 1) First, we lock **HONG_JI.ji_in** for read on a reserved RowHash to prevent a global deadlock.

- 2) Next, we do an all-AMPs SUM step to aggregate from **HONG_JI.ji_in** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.03 seconds.
 - 3) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 2 rows. The estimated time for this step is 0.04 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Example: Using a Join Index That Has an Extra Inner Join In Its Definition

The following example illustrates how the Optimizer uses a join index with an extra inner join when the connections among the tables in its definition are appropriately defined.

Suppose you have the following table definitions:

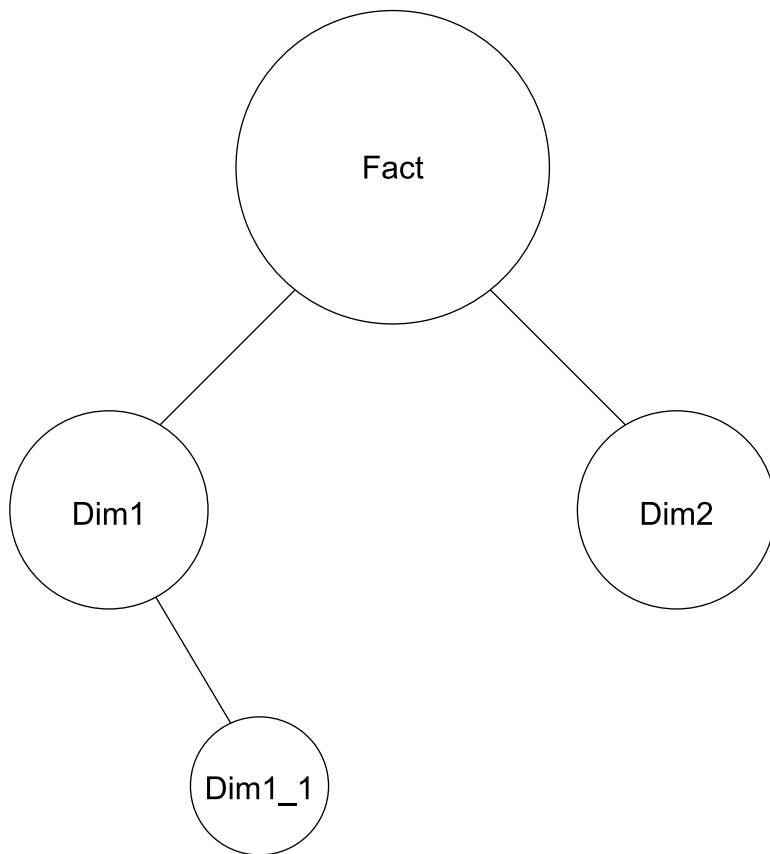
```
CREATE SET TABLE fact (
  f_d1 INTEGER NOT NULL,
  f_d2 INTEGER NOT NULL,
  FOREIGN KEY (f_d1) REFERENCES WITH NO CHECK OPTION dim1 (d1),
  FOREIGN KEY (f_d2) REFERENCES WITH NO CHECK OPTION dim2 (d2))
UNIQUE PRIMARY INDEX (f_d1,f_d2);

CREATE SET TABLE dim1 (
  a1 INTEGER NOT NULL,
  d1 INTEGER NOT NULL,
  FOREIGN KEY (a1) REFERENCES WITH NO CHECK OPTION dim1_1 (d11))
UNIQUE PRIMARY INDEX (d1);

CREATE SET TABLE dim2 (
  d1 INTEGER NOT NULL,
  d2 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (d2);

CREATE SET TABLE dim1_1 (
  d11 INTEGER NOT NULL,
  d22 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (d11);
```

The following graphic sketches the dimensional relationships among these tables:



In the following join index definition, the fact table is joined with dimension tables dim1 and dim2 by foreign key-primary key joins on fact.f_d1=dim1.d1 and fact.f_d2=dim2.d2. Dimension table dim1 is also joined with its dimension subtable dim1_1 by a foreign key-primary key join on dim1.a1=dim1_1.d11. A query on the fact and dim2 tables uses the join index ji_all because of its use of foreign key-primary key relationships among the tables:

```

CREATE JOIN INDEX ji_all AS
  SELECT (COUNT(*) (FLOAT)) AS countstar, dim1.d1, dim1_1.d11,
         dim2.d2, (SUM(fact.f_d1) (FLOAT)) AS sum_f1
  FROM fact, dim1, dim2, dim1_1
 WHERE ((fact.f_d1 = dim1.d1 )
        AND  (fact.f_d2 = dim2.d2 ))
        AND  (dim1.a1 = dim1_1.d11 )
  GROUP BY dim1.d1, dim1_1.d11, dim2.d2
  PRIMARY INDEX (d1);

```

Note that the results of the aggregate operations COUNT and SUM are both typed as FLOAT (see *Restrictions on Join Index Aggregate Functions* below).

As the bold text in the following EXPLAIN report indicates, the Optimizer uses the join index ji_all for its query plan in this situation because even though table fact is the parent table of both tables dim1 and dim2,

those tables are joined with fact on foreign key columns in the join index definition. Similarly, dim1 is joined to dim1_1 in the join index definition on a foreign key column.

```
EXPLAIN SELECT d2, SUM(f_d1)
          FROM fact, dim2
          WHERE f_d2=d2
          GROUP BY 1;
*** Help information returned. 18 rows.
*** Total elapsed time was 1 second.
```

Explanation

- 1) First, we lock **HONG_JI.ji_all** for read on a reserved RowHash to prevent a global deadlock.
 - 2) Next, we do an all-AMPS SUM step to aggregate from **HONG_JI.ji_all** by way of an all-rows scan with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed globally, then placed in Spool 3. The size of Spool 3 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.08 seconds.
 - 3) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPS. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.07 seconds.
 - 4) Finally, we send out an END TRANSACTION step to all AMPS involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

Examples of Exceptions to the General Rules for Extra Tables in the Join Index Definition

The following examples illustrate cases that are exceptions to the general coverage rules for extra tables in a join index definition.

Example: Table T9 is the Parent Table of Tables T7 and T8.

In the following example, table t9 is the parent table of tables t7 and t8. Generally, this relationship disqualifies a join index from covering any query with fewer tables than are referenced in the definition for that index. However, because t7 and t8 are joined on the FK columns (y7=x8) in the join index definition, the Optimizer uses the index ji to cover the query, as you can see by looking at the bold text in the EXPLAIN report:

```
CREATE SET TABLE t7(
  x7 INTEGER NOT NULL,
  y7 INTEGER NOT NULL,
  z7 INTEGER NOT NULL,
  CONSTRAINT r7 FOREIGN KEY (y7)
    REFERENCES WITH NO CHECK OPTION t9 (y9))
PRIMARY INDEX (x7);

CREATE SET TABLE t8(
  x8 INTEGER NOT NULL,
  y8 INTEGER NOT NULL,
  z8 INTEGER NOT NULL,
  CONSTRAINT r8 FOREIGN KEY (x8)
    REFERENCES WITH NO CHECK OPTION t9 (x9));

CREATE SET TABLE t9(
```

```

    x9 INTEGER NOT NULL UNIQUE,
    y9 INTEGER NOT NULL,
    z9 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX(y9);

CREATE JOIN INDEX ji AS
  SELECT x7, y7, x8, y8, x9, y9
  FROM t7, t8, t9
  WHERE y7=x8
  AND   y7=y9
  AND   x8=x9;

EXPLAIN SELECT x7, y7, x8, y8
  FROM t7, t8
  WHERE y7=x8
  AND   x7>1;

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

```

Explanation

- 1) First, we lock **HONG_JI.ji** for read on a reserved RowHash to prevent a global deadlock.
 - 2) Next, we do an all-AMPs RETRIEVE step from **HONG_JI.ji** by way of an all-rows scan with a condition of ("HONG_JI.ji.x1 > 1") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.06 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.06 seconds.

Example: SELECT Join Index Covers a Query

As with *Example: Table T9 is the Parent Table of Tables T7 and T8*, this example demonstrates how a join index can be used to cover a query when one table in the join index definition is a parent of two others if the tables are joined on foreign key-primary key relationships. t9 is the parent table of both t7 and t10. But because t7 and t10 are joined with t9 on the same PK column, by transitive closure, t7 and t10 are joined on y7=x10. The Optimizer does select join index ji to cover the query, as the bold text in the EXPLAIN report for the example query demonstrates:

```

CREATE SET TABLE t10(
  x10 INTEGER NOT NULL,
  y10 INTEGER NOT NULL,
  z10 INTEGER NOT NULL,
  CONSTRAINT r10 FOREIGN KEY ( x10 )
  REFERENCES WITH NO CHECK OPTION t9 ( y9 ))
PRIMARY INDEX x10;
CREATE JOIN INDEX ji AS
  SELECT x7, y7, x10, y10, x9, y9
  FROM t7, t10, t9
  WHERE y7=y9
  AND   x10=y9;
EXPLAIN SELECT x7, y7, x10, y10
  FROM t7, t10
  WHERE y7=x10
  AND   x7>1;

*** Help information returned. 14 rows.
*** Total elapsed time was 1 second.

```

Explanation

- 1) First, we lock **HONG_JI.ji** for read on a reserved RowHash to prevent a global deadlock.
 - 2) Next, we do an all-AMPs RETRIEVE step from **HONG_JI.ji** by way of an all-rows scan with a condition of ("**HONG_JI.ji.x1** > 1") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 3 rows. The estimated time for this step is 0.06 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.06 seconds.

Restrictions on Join Index Aggregate Functions

The following restrictions apply to aggregate functions used to define aggregate join indexes:

- Only the COUNT and SUM functions, in any combination, are valid.
- COUNT DISTINCT and SUM DISTINCT are not valid.
- To avoid overflow problems, always type the COUNT and SUM columns in a join index definition as FLOAT or DECIMAL(38,0) when the data type of the base table column is INTEGER or FLOAT, and as DECIMAL(38,xx) when the data type of the base table column is DECIMAL(NN,xx).

IF you ...	THEN the system ...
do not define an explicit data type for a COUNT or SUM column that has a data type of INTEGER or FLOAT	assigns the FLOAT data type to it.
do not define an explicit data type for a COUNT or SUM column that has a data type of DECIMAL	assigns the DECIMAL(38,xx) data type to it
define a COUNT or SUM column that has a data type of INTEGER as anything other than FLOAT and DECIMAL(38,0)	returns an error and does not create the aggregate join index.
define a COUNT or SUM column that has a data type of DECIMAL as anything other than DECIMAL(38,xx)	returns an error and does not create the aggregate join index.
define a COUNT or SUM column that has a data type of FLOAT as anything other than FLOAT	returns an error and does not create the aggregate join index.

Restrictions on Sparse Join Index WHERE Clause Predicates

The following restrictions apply to WHERE clause join conditions specified in a join index definition:

- Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.
- Multiple join conditions must be connected using the AND logical operator.
The OR operator is not a valid way to connect multiple join conditions in a join index definition.
- You cannot specify independent inequality WHERE clause join conditions in a join index definition.

The following rules apply to the use of inequality join conditions in a join index definition WHERE clause:

- Inequality join conditions are supported only if they are ANDed to at least one equality join condition.
- Inequality join conditions can be specified only for columns having the same data type in order to enforce domain integrity.
- The only valid comparison operators for an inequality join condition are the following:

<

<=

>

>=

The following join index definition is valid because the WHERE clause inequality join condition on o_totalprice and c_acctbal it specifies is ANDed with the previous equality join condition on o_custkey and c_custkey:

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT c_name, o_orderkey, o_orderdate
  FROM orders, customer
  WHERE o_custkey = c_custkey
  AND   o_totalprice > c_acctbal;
```

The following join index definition is not valid because the WHERE clause inequality join condition has no logical AND relationship with an equality join condition:

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT c_name, o_orderkey, o_orderdate
  FROM orders, customer
  WHERE o_totalprice > c_acctbal;
```

Restrictions on Join Index ORDER BY Clauses

The following restrictions apply to ORDER BY clauses specified in a join index definition:

- Sort order is restricted to ASC.
DESC is not valid.
- Aggregate columns and expressions are not permitted as a *column_name* or *column_position* specification.
- Supported data types for *column_name* are restricted to the following:
 - DATE
 - BYTEINT
 - DECIMAL

The DECIMAL type is valid only for columns of four or fewer bytes.

- INTEGER
- SMALLINT

Restrictions on Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have join indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with join indexes using these utilities, the load operation aborts and the system returns an error message to the requestor.

To load data into a join-indexed base table, you must drop all defined join indexes on that base table before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Be aware that you cannot drop a join index to enable MultiLoad or FastLoad batch loads until any requests that access that index complete processing. Requests place locks on any join indexes they access, and the system defers processing of any DROP JOIN INDEX requests against locked indexes until their locks have all been released.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, are supported for join-indexed tables, as are multirow INSERT SELECT and MERGE requests.

Improving Join Index Performance

Depending on the complexity of the joins or the cardinalities of base tables that must be aggregated, join indexes optimize the performance of certain types of workloads.

Selecting the Primary Index for Join Indexes

As with any other Vantage table, you must define a primary index for any join index you create unless it is column-partitioned (in which case it is optional). While the primary index for any join index must be defined as a NUPI, the column on which it is defined need not be nonunique; in fact, the usual preferences for defining a unique rather than a nonunique column set as the primary index for a base table also apply to join indexes.

Because join indexes are not part of the logical model for a database, and because they are de facto denormalized database objects, there might not be an overriding reason to define one column set over another as the primary index apart from how much more evenly one set might distribute join index table rows than another.

It is usually important to use the join key as the primary index for single-table join indexes, but otherwise there is no compelling theoretical reason to select one unique (or, if not unique, highly singular) column set over another as the primary index. If you create a join index to support a row-partitioned base table, you should consider creating it using row partitioning. This is valid only if the join index is not also row-compressed. You can also create a row-partitioned join index for a base table that does not have row partitioning. It is always possible that such a join index might provide an alternative organization of the data that facilitates access based on partitions. For example, suppose you have a nonpartitioned base table designed to handle efficient joins on its primary index. You might also want to create a row-partitioned

join index on the table that optimizes fast row-partition-based access to the data. The rules for creating a row-partitioned join index are generally the same as those for creating a row-partitioned base table.

If a join index is designed to support range queries, you should consider specifying a row-partitioned primary index for it. Note that you cannot specify a row-partitioned primary index for a row-compressed join index.

Join index primary indexes are defined analogously to base table primary indexes: with the CREATE JOIN INDEX statement.

Selecting Secondary Indexes for Join Indexes

Join indexes are often more high-performing if they have one or more secondary indexes defined on them. The Optimizer adds a join index (even a partially covering join index) to its query plan whenever it can, and just because you define it to have a particular most likely use and access path, there is no reason to believe that the same join index might not also be useful for other, unplanned, queries. The Optimizer will join a base table that is unrelated to a join index with that join index if the query plan can be made more cost effective by doing so.

A secondary index on a join index cannot be defined as UNIQUE, even though the column set on which it is defined is unique. This rule is enforced because of the way indexes on join index tables are handled internally.

For further information about creating join indexes and using secondary indexes with them, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Statistics and Other Demographic Data for Join Indexes

Collect statistics on the indexes of a join index to provide the Optimizer with the information it needs to generate an optimal plan.

The COLLECT STATISTICS (Optimizer Form) statement collects demographics, computes statistics from them, and writes the resulting data into individual entries for each individual base table and join index table on the system.

As far as the Optimizer is concerned, a multitable join index and the base tables it supports are entirely separate entities. You must collect statistics on multitable join index columns separately from the statistics you collect on their underlying base table columns because the column statistics for multitable join indexes and their underlying base tables are not interchangeable.

On the other hand, it is generally preferable to collect statistics on the underlying base table of a single-table join index and not directly on the join index columns. Note that the derived statistics framework supports bidirectional inheritance of statistics between a non-sparse single-table join index and its underlying base table. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Guidelines for Collecting Statistics on Multitable Join Indexes

The guidelines for collecting statistics on the relevant columns are the same as those for any regular join query that is frequently executed or whose performance is critical. The only difference with join indexes is that the join result is persistently stored and maintained by the AMP software without user intervention.

Note the following guidelines for collecting statistics for join indexes.

- To improve the performance of creating a join index and maintaining it during updates, collect statistics on its base tables immediately prior to creating the join index.
- Collect statistics on all the indexes defined on your join indexes to provide the Optimizer with the information it needs to generate an optimal plan.
- Collect statistics on additional join index columns that frequently appear in WHERE clause search conditions, especially when the column is the sort key for a value-ordered join index because the Optimizer can then use that information to more accurately compare the cost of using a NUSI-based access path in conjunction with range or equality conditions specified on the sort key column.
- In general, there is no benefit in collecting statistics on a join index for joining columns specified in the join index definition itself. Statistics related to these columns should be collected on the underlying base tables rather than on the join index.

The only time you gain an advantage by collecting statistics on a join column of the join index definition is when that column is used as a join column to other base tables in queries where the join index is expected to be used in the Optimizer query plan.

Guidelines for Collecting Statistics on Single-Table Join Indexes

For most applications, you should collect the statistics on base table columns rather than on single-table join index columns.

You can collect and drop statistics on the primary index columns of a single-table join index using an INDEX clause to specify the column set. For example, suppose a join index has the following definition.

```
CREATE JOIN INDEX OrdJIdx8 AS
SELECT o_custkey, o_orderdate
FROM Orders
PRIMARY INDEX (o_custkey, o_orderdate)
ORDER BY (o_orderdate);
```

Then you can collect statistics on the index columns as a single object as shown in the following example.

```
COLLECT STATISTICS ON OrdJIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can only use columns specified in a PRIMARY INDEX clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_custkey);
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like `WHERE x=1 AND y=2` is better optimized if statistics are collected on `INDEX (x,y)` than if they are collected individually on column `x` and column `y`.

The same restrictions hold for the `DROP STATISTICS` statement.

Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns

The Optimizer substitutes base table statistics for single-table join index statistics when no demographics have been collected for its single-table indexes. Because of the way single-table join index columns are built, it is generally best not to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

You might need to collect statistics on single-table join index columns instead of their underlying base table columns if you decide not to collect the statistics for the relevant base table columns for some reason. In this case, you should collect statistics directly on the corresponding single-table join index columns.

Note that the derived statistics framework can use bidirectional inheritance of statistics between base tables and their underlying non-sparse single-table join indexes, so the importance of collecting statistics on base table columns rather than non-sparse single-table join index columns is no longer as important as it once was (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details).

Guidelines for Collecting Statistics On Single-Table Join Index Columns

The guidelines for selecting single-table join index columns on which to collect statistics are similar to those for base tables. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce worse access plans, then you should always report the incident to Teradata Services.

When you are considering collecting statistics for a single-table join index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a single-table join index must access it with a direct probe, a full table scan, or a range scan. With this in mind, consider the following factors when deciding which columns to collect statistics for.

- Always consider collecting statistics on the primary index. This is particularly critical for accurate cardinality estimates.

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
joining the single-table index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

IF a single-table join index is defined ...	THEN you should collect statistics on the ...
with an ORDER BY clause	order key specified by that clause.
without an ORDER BY clause and the order key column set from the base table is not included in the <i>column_name_1</i> list	order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics.

- If a single-table join index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered single-table join index.

Join Index Storage

The storage organization for join indexes supports a row compressed format to reduce storage space.

If you know that a join index contains groups of rows with repeating information, then its definition DDL can specify repeating groups, indicating the repeating columns in parentheses. The column list is specified as two groups of columns, with each group stipulated within parentheses. The first group contains the non-repeating columns and the second group contains the repeating columns.

You can elect to store join indexes in value order, ordered by the values of a 4-byte column. Value-ordered storage provides better performance for queries that specify selection constraints on the value ordering column. For example, suppose a common task is to look up sales information by sales date. You can create a join index on the sales table and order it by sales date. The benefit is that queries that request sales by sales date only need to access those data blocks that contain the value or range of values that the queries specify.

Physical Join Index Row Compression

Compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

A physical join index row has:

- One set of values for the columns in the *column_1* list. When a *column_2* list is specified, the columns specified in the *column_1* list are the data that is compressed for rows that have the same values for the columns in the *column_1* list.
- Multiple sets of values of the columns in the *column_2* list for rows that have the same *column_1* list values.

For example, if a logical join result has n rows with the same fixed part value, then there is one corresponding physical join index row that includes n repeated parts in the physical join index. A physical join index row represents one or more logical join index rows. The number of logical join index rows represented in the physical row depends on how many repeated values are stored.

Compression is only done on rows inserted by the same INSERT statement. Newly inserted rows are not added as logical rows to existing compressed rows.

When the number of repeated values associated with a given fixed value exceeds the system row size limit, the join index row is automatically split into multiple physical rows, each having the same fixed value but different lists of repeated values. Note that a given logical join index result row cannot exceed the system 1 MB row size limit.

Guidelines for Using Row Compression With Join Index Columns

A column set with a high number of distinct values cannot be row compressed because it rarely (and in the case of a primary key, never) repeats. Other column sets, notably foreign key and status code columns, are generally highly non-distinct: their values repeat frequently. Row compression capitalizes on the redundancy of frequently repeating column values by storing them once in the fixed part of the index row along with multiple repeated values in the repeated part of the index row.

Typically, primary key table column values are specified as the repeating part and foreign key table columns are specified in the fixed part of the index definition.

[Using Outer Joins to Define Join Indexes](#) and [Creating Join Indexes Using Outer Joins](#) are examples of how you can use row compression to take advantage of repeated column values.

A row-compressed join index might be used for a query when the join index would be used if it were not row compressed. Check the EXPLAIN for a query to see if join indexes are being used.

Determining the Space Overhead for a Hash or Join Index

The ROWID, if included, is always 10 bytes long for hash indexes. For join indexes, the space overhead for the ROWID is 10 bytes for 2-byte partitioning and 16 bytes for 8-byte partitioning.

The following equation provides an estimate of the space overhead required for an uncompressed hash index. Double the result if you define fallback on the index.

Uncompressed hash or join index size = $N \times (F + O)$

where:

Symbol	Description
N	Cardinality of the base table.
F	Length of the columns, including base row identifier information.
O	Row overhead. Assume the following row overhead for a join index depending on whether it is partitioned or not: <ul style="list-style-type: none"> • 12 bytes for a nonpartitioned join index • 14 bytes for a join index with 2-byte partitioning • 20 bytes for a join index with 8-byte partitioning

The following equation provides an estimate of the space overhead required for a row-compressed hash or join index. Double the result if you define fallback on the index.

Row-compressed hash or join index size = $U \times (F + O + (R \times A))$

where:

Symbol	Description
F	Length of the fixed column <i>column_name</i> . For a hash index, this length includes the base row identifier information in the fixed column.
R	Length of a single repeating column <i>column_name</i> . For a hash index, this length includes the base row identifier information in the repeating column.
A	Average number of repeated columns for a given value in <i>column_1_name</i> .
U	Number of unique values in the specified <i>column_1_name</i>
O	Row overhead. Assume the following row overhead for a hash or join index depending on whether it is partitioned or not: <ul style="list-style-type: none"> • 12 bytes for a nonpartitioned hash or join index • 14 bytes for a join index with 2-byte partitioning • 20 bytes for a join index with 8-byte partitioning

Considerations for Measuring Disk Space

Keep the following rules of thumb in mind whenever you perform these calculations.

- Implement your measurements over a very large number of rows to avoid distorting the figures with table overhead issues.
- An INTEGER column uses 4 bytes.

A SUM(INTEGER) column uses 8 bytes because the system always casts it to FLOAT.

- If you have not explicitly specified a COUNT column in your aggregate join index definition, add 4 bytes to the definition to account for the required COUNT column.

Example Measurement 1

Table A and Table B both have 60 million 100-byte rows. An aggregate join index with two INTEGER columns and one SUM(INTEGER) column.

From these figures, the computed size of the aggregate join index is as follows:

Row size = $(2 \times 4 \text{ INTEGER bytes}) + 4 \text{ COUNT bytes} + 8 \text{ SUM bytes} + 14 \text{ overhead bytes} = 34 \text{ bytes}$

Type of Join Index	Join Index Row
Non-aggregate	For every matching row in the base table.
Aggregate	Per group of rows in the join of the base tables.

The permanent space specification is as follows:

Current Permanent Space (Bytes)	Peak Permanent Space (Bytes)	Maximum Permanent Space (Bytes)
2,041,595,904	2,041,595,904	0

Using these figures, you can compute a measured row size:

Row size = $2,041,595,904 \div 60,000,000 = 34.03$ bytes

Because the measured and computed disk spaces are identical within a narrow confidence interval, it is safe to conclude that the disk space formula is accurate for an aggregate join index.

Example Measurement 2

This example examines the size of a simple join index created for the same tables used in *Example Measurement 1* above. There is a savings of 8 bytes because there is no COUNT column and no SUM(INTEGER) column.

The computed row size is as follows:

Row size = 34 bytes – (4 INTEGER bytes + 4 SUM(INTEGER) bytes) = 26 bytes

The permanent space specification is as follows.

Current Permanent Space (Bytes)	Peak Permanent Space (Bytes)	Maximum Permanent Space (Bytes)
1,561,216,000	1,561,216,000	0

Using these figures, you can compute a measured row size.

Row size = $1,561,216,000 \div 60,000,000 = 26.02$

As with *Example Measurement 1*, the measured and computed disk spaces are identical, and you can conclude that the disk space formula is accurate for a simple join index.

Value-Ordered Storage of Join Index Rows

The distribution of the subtable rows for a join index across the AMPs is controlled by its NUPI. By default, join index subtable rows are sorted locally on each AMP by the row hash value of the NUPI column set. You can also specify that rows be stored in value-order.

Value Ordering and Range Queries

You can specify value ordering by means of the optional ORDER BY clause in the join index definition. Sorting a join index by values, as opposed to row hash values, is especially useful for range queries involving the sort key. Value ordering is limited to a single numeric or DATE columns with a size of 4 bytes or less.

For example, the following join index rows are hash-distributed using c_name and are value-ordered on each AMP using c_custkey as the sort key.

```

CREATE JOIN INDEX OrdCustIdx AS
  SELECT (o_custkey
         ,c_name)
        ,
        (o_status
         ,o_date
         ,o_comment)
FROM Orders LEFT JOIN Customer
ON o_custkey = c_custkey
ORDER BY o_custkey
PRIMARY INDEX (c_name);

```

Hash Indexes

Hash indexes are file structures that share properties in common with both single-table join indexes and secondary indexes.

From an architectural perspective, the incorporation of auxiliary structures as a transparently embedded element of the hash index column set is what most distinctly distinguishes hash indexes from single-table join indexes. These auxiliary structures are components of the base table, and are added to the hash index definition by default if they are not explicitly declared by the CREATE HASH INDEX column set definition. Because it is not clear which default auxiliary structures Vantage uses when it create a hash index, you should always consider creating an equivalent single-table join index in preference to a hash index. Also, multivalue compression from the base table may be carried over to a join index, but it is not carried over to a hash index.

If the columns you specify for the hash index column set duplicate the default auxiliary structure columns, then those columns are not added redundantly. The auxiliary structures provide indexed access to base table rows.

If you do not specify a partition key explicitly with the BY clause of the CREATE HASH INDEX statement, then the system adds this auxiliary pointer data to the hash index rows automatically and then uses it to partition them.

Comparison of Hash Indexes and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Not only can hash indexes optionally be specified to be distributed in such a way that their rows are AMP-local with their associated base table rows, they can also provide a transparent direct access path to those base table rows to complete a query only partially covered by the index. This facility makes hash indexes somewhat similar to secondary indexes in function. Hash indexes are also useful for covering queries so that the base table need not be accessed at all.

The most apparent external difference between hash and single-table join indexes is in the syntax of the SQL statements used to create them. The syntax for CREATE HASH INDEX is similar to that for CREATE

INDEX. As a result, it is simpler to create a hash index than to create a functionally comparable single-table join index.

The following list summarizes the similarities of hash and single-table join indexes.

- The primary function of both is to improve query performance.
- Both are maintained automatically by the system when the relevant columns of their base table are updated by a DELETE, INSERT, or UPDATE statement.
- Both can be the object of any of the following SQL statements.
 - COLLECT STATISTICS (Optimizer Form)
 - DROP STATISTICS
 - HELP INDEX
 - SHOW HASH INDEX
- Both receive their space allocation from permanent space and are stored in distinct tables.
- Both can be hash- or value-ordered. You must drop and rebuild all value-ordered (but not hash-ordered) hash and join indexes after a system has been reconfigured. For more information on system reconfigurations and the Reconfiguration utility, contact the Teradata Support Center.
- Both can be row compressed.
- Both can be FALLBACK protected.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

- Both can be used to transform a complex expression into a simple index column. This transformation permits you to collect statistics on the expression, which the Optimizer can then use to make single-table cardinality estimates for a matching complex column predicate specified on the base table and for mapping a query expression that is identical to an expression defined in the join index, but is found within a non-matching predicate (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for details).
- Neither can be queried or directly updated.
- A hash index cannot have a partitioned primary index, but a single-table join index can.
- A hash index must have a primary index, but a single-table join index can be created with or without a primary index if it is column-partitioned.
- A hash index cannot be column-partitioned, but a single-table join index can be column-partitioned.
- Neither can be used to partially cover a query that contains a TOP *n* or TOP *m* PERCENT clause.
- Neither can be implemented with row compression if they specify a UDT in their select list because both create an internal column1 and column2 index when compressed.
- Neither can be defined using the system-derived PARTITION column.
- Both share the same restrictions for use with the MultiLoad and FastLoad utilities.

The following table summarizes the important differences between hash and join indexes.

Hash Index	Join Index
Indexes one table only.	Can index multiple tables. This is not true for column-partitioned join indexes, which can only index a single table.
A logical row corresponds to one and only one row in its referenced base table.	A logical row can correspond to either of the following, depending on how the join index is defined: <ul style="list-style-type: none"> • One and only one row in the referenced base table. • Multiple rows in the referenced base tables.
Column list cannot specify aggregate or ordered analytical functions.	Select list can specify aggregate functions.
Can specify a UDT column in its column list.	Can only specify a UDT in its select list if it is not row-compressed.
Cannot have a nonunique secondary index.	Can have a nonunique secondary index.
Supports transparently added, system-defined primary index columns that point to the underlying base table rows.	Does not add underlying base table row pointers implicitly. Pointers to underlying base table rows can be created explicitly by defining one element of the column list using the keyword ROWID. You can specify ROWID only in the outermost SELECT of the CREATE JOIN INDEX statement (see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144).
Cannot be specified for NoPI or column-partitioned base tables.	Can be specified for both NoPI and column-partitioned base tables.
Cannot be column partitioned.	Can be column partitioned.
Cannot be row partitioned.	Primary index of uncompressed row forms and column-partitioned join indexes, can be row partitioned.
Cannot be defined on a table that also has triggers.	Can be defined on a table that also has triggers.
Column multivalue compression, if defined on a referenced base table, is not added transparently by the system and cannot be specified explicitly in the hash index definition.	Column multivalue compression, if defined on a referenced base table, is added transparently by the system with no user input, but cannot be specified explicitly in the join index definition.
Index row compression is added transparently by the system with no user input.	Index row compression, if used, must be specified explicitly in the CREATE JOIN INDEX request by the user.

It is possible to define a join index that has nearly the identical functionality to a hash index. The only essential differences between hash and join indexes is their respective DDL creation syntax. Otherwise, the functionality of hash indexes is a proper subset of the functionality of join indexes.

Summary of Hash Index Functions

A hash index always has at least one of the following functions.

- Replicates all, or a vertical subset, of a single base table and partitions its rows with a user-specified partition key column set, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Provides an access path to base table rows to complete partial covers.

The AMP software updates hash indexes automatically, so the only task a DBA must perform is to keep the statistics on hash index or base table columns current (see [Collecting Statistics on Hash Index Columns](#)).

Similarities of Hash Indexes to Base Tables

In many respects, a hash index is identical to a base table.

For example, you perform any of the following statements against a hash index:

- COLLECT STATISTICS (Optimizer Form)
- DROP HASH INDEX
- DROP STATISTICS (Optimizer Form)
- HELP HASH INDEX
- SHOW HASH INDEX

You cannot do the following things with hash indexes:

- Query or update hash index rows.

For example, if ordCustHdx is a hash index, then the following query is not legal:

```
SELECT o_status, o_date, o_comment
FROM ordCustHdx;
```

- Store and maintain arbitrary query results.
- Create secondary indexes on its columns.
- Create them on either a NoPI or column-partitioned table.

Similarities of Hash Indexes to Secondary Indexes

Hash indexes are file structures that can be used either to resolve queries by accessing the index instead of its underlying base table or to enhance access performance when they do not cover a query by providing a secondary access path to requested base table rows. They can either substitute for or point to base table rows.

Because of these properties, hash indexes are useful for queries where the index structure contains all the columns referenced by a query, thereby *covering* the query, and making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index.

Hash indexes put in double duty by also providing pointers to base table rows to facilitate their access in situations where the index does not cover the query. This application of hash indexes is similar to how the Optimizer uses secondary indexes when it creates its access plans.

Because the distribution of rows to AMPs of hash index rows is under user control through the specification of an explicit partition key in the CREATE HASH INDEX statement, rows can be distributed in various ways, depending on the requirements of an application. Distribution of secondary index rows, on the other hand, is not under user control. Because of this, they are less adaptable to the specific requirements of an individual application than hash indexes.

Both secondary and hash indexes provide access paths to base table rows and can be selected by the Optimizer to cover SQL queries. Both also share a similar DDL syntax that is very different from the syntax used to create single-table join indexes.

The key difference between hash and secondary indexes is that the hash index partition key is user-selectable, which often makes it more useful for processing queries.

Hash Index Applications

Hash indexes are useful for queries where the index table contains the columns referenced by a query, thereby allowing the Optimizer to cover it by planning to access the index rather than its underlying base table. An index that supplies all the table columns requested by a query is said to cover that table for that query and, for obvious reasons, is referred to as a covering index. Note that query covering is not restricted to hash indexes: other indexes can also cover queries.

Hash indexes can be defined on a table in place of secondary indexes. This usage makes more sense when you distribute the hash index row to the AMPs so that it facilitates a wider range of query processing than a secondary index might. Because hash indexes can potentially carry more of an update burden than secondary indexes, you should not define them on a table when a secondary index would serve the same intended function. Keep in mind that this depends on how the updates are done. For example, a single row update might be faster with a secondary index, but an INSERT SELECT might be faster with a hash index.

Compression of Hash Index Rows

Compression refers to a logical row compression in which multiple sets of non-repeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any non-repeating column values are stored as logical segmental extensions of the base repeating set.

When the following is true, the system automatically compresses the rows of a hash index:

- The ORDER BY column set is specified in the *column_1* list and none of the columns in the order key is unique.
- The primary index columns are specified in the *column_1* list.

As the following table indicates, the primary index columns for a hash index are always part of the *column_1* list, whether specified explicitly or not.

IF you create the index ...	THEN the primary index columns ...
without a BY clause	are part of the <i>column_1</i> list by default.
with a BY clause	must be part of the <i>column_1</i> list because all columns specified in the BY clause column list must also be specified in the <i>column_1</i> list.

Rows having the same values for the order key are compressed into a single physical row having fixed and repeating parts. If the columns do not fit into a single physical row, they spill over to additional physical rows as is necessary.

The fixed part of the row is made up of the explicitly-defined columns that define the *column_1* list. The repeating part is composed of the remaining, implicitly-defined columns.

The system only compresses row sets together if they are inserted by the same INSERT statement. This means that rows that are subsequently inserted are not appended as logical rows to existing compressed row sets, but rather are compressed into their own self-contained row sets.

Fallback With Hash Indexes

You can define fallback for hash indexes. The criteria for deciding whether to define a hash index with fallback are similar to those used for deciding whether to define fallback on base tables.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

If you do not define fallback for a hash index and an AMP is down, then the following additional factors become critical.

- The hash index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the hash index is defined cannot be updated.

Restrictions for NoPI Tables

You cannot create a hash index on a NoPI or a column-partitioned table because hash indexes are defined with transparently added, system-defined primary index columns from the underlying base table primary index that point to the underlying base table rows, and NoPI tables do not have a primary index.

Restrictions on Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have hash indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with hash indexes using these utilities, the load operation aborts and the system returns an error message to the requestor.

To load data into a hash-indexed base table, you must drop all defined hash indexes on the table before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like Teradata Parallel Data Pump, BTEQ, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, are supported for hash-indexed tables.

Restriction on Partial Coverage of Queries Containing a TOP *n* or TOP *m* PERCENT Clause

A hash index cannot be used to partially cover a query that specifies the TOP *n* or TOP *m* PERCENT option.

Compression of Hash Indexes at the Block Level

Because hash indexes are primary tables, they can be compressed at the block level. See [Compression Types Supported by Vantage](#) for more information about data block compression.

Further Information

Consult *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more detailed information on creating and using hash indexes to enhance the performance of your database applications.

Collecting Statistics on Hash Index Columns

When there are no statistics for a hash index, the Optimizer uses the statistics of the corresponding base table rows, just as it does for a single-table join index.

The guidelines for selecting hash index columns on which to collect statistics are similar to those for base tables and join indexes. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce worse access plans, you should always report the incident to Teradata Services personnel.

When you consider collecting statistics for a hash index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a hash index must access it with a direct probe, a full table scan, or a range scan.

With this in mind, consider the following when deciding which columns to collect statistics for.

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
joining the hash index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

IF a hash index is defined ...	THEN you should collect statistics on the ...
with a BY clause or ORDER BY clause (or both)	primary index and order keys specified by those clauses.

IF a hash index is defined ...	THEN you should collect statistics on the ...
without a BY clause	primary index column set of the base table on which the index is defined.
without an ORDER BY clause and the order key column set from the base table is not included in the <i>column_name_1</i> list	order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics, including the cardinality of the hash index.

- If a hash index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered hash index.
- Consider collecting statistics on the base table columns that are also part of the hash index rather than collecting statistics on the hash index itself.

General Guidelines

You should collect statistics on appropriate columns of a hash index frequently just as you would for any base table or join index.

Note:

For most applications, you should collect the statistics on base table columns rather than on hash index columns.

When you create a hash index with a BY clause, you can collect and drop statistics on those columns using an INDEX clause to specify the column set. For example, suppose a hash index has the following definition.

```
CREATE HASH INDEX OrdHIdx8 (o_custkey, o_orderdate) ON orders
BY (o_custkey, o_orderdate)
ORDER BY (o_orderdate);
```

Then you can collect statistics on the partitioning columns as shown in the following example.

```
COLLECT STATISTICS ON OrdHIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can only use columns specified in a BY clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_custkey);

COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like `WHERE x=1 AND y=2` is better optimized if statistics are collected on `INDEX (x,y)` than if they are collected individually on column `x` and column `y`.

The same restrictions hold for the `DROP STATISTICS` statement.

Collecting Statistics on Base Table Columns Instead of Hash Index Columns

The Optimizer substitutes base table statistics for hash index statistics when no demographics have been collected for its hash indexes. Because of the way hash index columns are built, it is generally best not to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

Collect statistics on a hash index or non-sparse join index directly in the following scenarios:

- If you decide not to collect the statistics for the relevant base table columns for some reason, then you should collect them directly on the corresponding hash index columns.
- If the hash index is row or column partitioned, collect single-column and multicolumn `PARTITION` statistics directly on the hash index.
- Collect `SUMMARY` statistics directly on the hash index as the summary attributes such as database block size, row size, etc. differ from the underlying base table.

Hash Index Definition Restrictions

This topic lists several restrictions on hash index definitions. For complete details about hash index syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Restrictions for NoPI Tables

You cannot create a hash index on a NoPI table.

Restrictions on Number of Hash Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table is 32, in any combination. This includes the system-defined unique secondary indexes used to implement `PRIMARY KEY` and `UNIQUE` constraints.

Each composite NUSI that specifies an `ORDER BY` clause counts as 2 consecutive indexes in this calculation (see [Value-Ordered NUSIs and Range Conditions](#)). You cannot define hash, or any other, indexes on global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184).

For example, suppose you have 4 tables, each with multiple secondary, hash, and join indexes defined on them.

- `table_1` has 32 secondary indexes and no hash or join indexes.

- table_2 has 16 secondary indexes, no hash indexes, and 16 join indexes.
- table_3 has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- table_4 has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but they all operate at the boundaries of the defined limits.

Note that if any of the secondary indexes defined on tables 1, 2, or 3 is a NUPI defined with an ORDER BY clause, the defined limits are exceeded, and the last index you attempt to create on the table will fail. Because each NUPI defined with an ORDER BY clause counts as 2 consecutive indexes in the count against the maximum of 32 per table, you could define only 8 of them on table_2, for example, if you also defined 16 join indexes on the table.

Restrictions on Number of Columns Per Referenced Base Table

The maximum number of columns that can be specified in a hash index is 64.

Restrictions on the Use of the System-Derived PARTITION[#L n] Column in a Hash Index Definition

You cannot use the system-derived PARTITION[#L n] column in the definition of a hash index.

Restriction on Number of Hash Indexes Selected Per Base Table

The Optimizer can use several hash indexes for a single query, selecting one or more multitable join indexes as well as additional hash indexes for its join plan. The hash indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable hash indexes for the plan. Always examine your EXPLAIN reports to determine which hash indexes are used for the query plans generated for your queries. If a hash index you think should have been used by a query was not included in the query plan, try restructuring the query and then EXPLAIN it once again.

The join planning process selects a multitable join index to replace any individual table in a query when the substitution further optimizes the query plan. For each such table replaced in the join plan by a multitable join index, as many as two additional hash indexes can also be added if their inclusion reduces the size of the relation to be processed, provides a better distribution, or offers additional covering.

The limit on the number of hash indexes substituted per individual table in a query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort: in other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

Hash and Join Index Interactions With Other Teradata Systems and Features

This topic describes how hash and join indexes interact with the following Teradata systems and features.

Summary

The following table summarizes the Vantage features that do not support hash or join indexes.

Feature	Reason Not Supported	Recommended Workaround
Triggers	Triggers are handled by the Resolver, while hash indexes are handled by the Optimizer.	Do not define triggers and hash indexes on the same base tables.
Permanent journal recovery	Recovery process does not rebuild hash and join indexes.	Rebuild hash and join indexes after the permanent journal recovery completes.
MultiLoad	Utilities do not maintain hash and join indexes.	<ol style="list-style-type: none"> 1. Ensure that no queries are running against tables that use the hash or join index to be dropped. 2. Drop hash and join indexes before loading or restoring the base tables. 3. Recreate hash and join indexes after loading or restoring the base tables.
FastLoad		
Teradata Parallel Transporter LOAD and UPDATE operators		
<ul style="list-style-type: none"> • NoPI tables • Column-partitioned tables 	Hash indexes automatically add the primary index of their underlying base tables to their definition, and NoPI tables have no primary index.	None.

Triggers

You cannot define triggers and hash indexes on the same table; however, you can define triggers and join indexes on the same tab

Whichever feature is defined first for a table blocks the other from being created and returns an error message to the requestor.

Permanent Journal Recovery

You can use ROLLBACK or ROLLFORWARD utility commands to recover base tables with hash or join indexes defined on them; however, the indexes are not rebuilt during the recovery process.

Instead, any such hash or join index is marked as non-valid, and you must drop and recreate it before the Optimizer can use it again in a query plan.

When a hash or join index has been marked not valid, the SHOW HASH INDEX and SHOW JOIN INDEX statements display a special status message to inform you that the index has been so marked.

Load Utilities

You cannot use FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE to load data into base tables that have hash or join indexes because those indexes are not maintained during the execution of these utilities (see *Teradata® FastLoad Reference*, B035-2411, *Teradata® MultiLoad Reference*, B035-2409, and *Teradata® Parallel Transporter Reference*, B035-2436 for details).

If you attempt to load data into base tables with hash or join indexes using these utilities, an error message returns and the load does not continue.

To load data into hash- or join-indexed base table, you must drop all defined hash or join indexes before you can run FastLoad, MultiLoad, or the Teradata Parallel Transporter operators LOAD and UPDATE.

Load utilities like BTEQ, Teradata Parallel Data Pump, and the Teradata Parallel Transporter operators INSERT and STREAM, which perform standard SQL row inserts and updates, are supported for hash- and join-indexed tables (see *Basic Teradata® Query Reference*, B035-2414, *Teradata® Parallel Data Pump Reference*, B035-3021, and *Teradata® Parallel Transporter Reference*, B035-2436 for details).

You cannot drop a hash join or index to enable batch data loading by utilities such as MultiLoad and FastLoad as long as queries are running that access that index. Each such query places a lock on the index while it is running, so it blocks the completion of any DROP JOIN INDEX or DROP HASH INDEX transactions until the lock is removed.

Furthermore, as long as a DROP JOIN INDEX or DROP HASH INDEX transaction is running, batch data loading jobs against the underlying tables of the index cannot begin processing because of the EXCLUSIVE locks DROP JOIN INDEX and DROP HASH INDEX place on the base table set that defines them.

Teradata Parallel Data Pump

You can use the Teradata Parallel Data Pump utility, which performs standard SQL row inserts and updates, to load data into base tables that have a hash or join index defined on them because those indexes are properly maintained during its execution. For more information, see *Teradata® Parallel Data Pump Reference*, B035-3021.

Tradeoffs for Join or Hash Indexes

Join and hash indexes can also have properties that make other solutions more optimal. Remember: it is always important to test and validate join and hash indexes on a test system before putting them into a production environment. Keep these potential downside characteristics in mind whenever you consider creating a join or hash index that you think might be useful for your application workload.

- Balance usage broadness with access efficiency. The more columns defined for a join or hash index, the longer the access time for processing the index. The more tables included in a hash or join index definition, the longer and more burdensome the required maintenance time.
- You cannot define row partitioning for a hash index, nor can you define row or column partitioning for a row-compressed join index.

However, you can define row or column partitioning for an uncompressed join index.

Note that you cannot collect statistics on the PARTITION column of such a join index.

- You cannot use the MultiLoad, FastLoad, or Restore utilities against tables that have join or hash indexes defined on them. See [Load Utilities](#), *Teradata® FastLoad Reference*, B035-2411, and *Teradata® MultiLoad Reference*, B035-2409 for further information.

For many bulk loading applications, you can instead FastLoad rows into a staging table which you then MERGE (using error logging) into the target table (for details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

- You cannot define triggers on tables that have hash indexes defined on them; however, you can define join indexes and triggers on the same table. See [Triggers](#).

Designing for Database Integrity

This section focuses on two important database design issues related to the integrity of databases: semantic and physical data integrity. First and foremost, enforcing database integrity means getting the correct results. This is fundamental.

The quality of data that is received from different sources can vary immensely. Efforts are currently underway to develop tools for data cleaning that allow users to estimate the reliability of uncertain data using Bayesian statistical methods, which can be used both to develop probability models for uncertain data and to improve the quality of uncertain data.

Sources of Data Quality Problems

Data Warehousing Institute 2002 Survey

The following table reports the results of an industry survey of sources of data quality problems.

Source	Percentage
Data entry by employees	76
Changes to source systems	53
Data migration or conversion projects	48
Mixed expectations by users	46
External data	34
System errors	26
Data entry by customers This includes typographical errors and information typed into the wrong fields made when entering data into forms on the World Wide Web.	25
Other	12
Source: Wayne W. Eckerson, <i>Data Quality and the Bottom Line: Achieving Business Success Through a Commitment to High Quality Data</i> , The Data Warehousing Institute, 2002.	

Data Warehousing Institute 2005 Follow-Up Survey

Three years later, a TDWI survey of many of the same industry sources reported the following list of most frequent contributors to data quality problems in their organizations:

Source	Percentage
Data entry by employees	75

Source	Percentage
Inconsistent definitions for common terms	75
Data migration or conversion projects	46
Mixed expectations by users	40
External data	38
Data entry by customers This includes typographical errors and information typed into the wrong fields made when entering data into forms on the World Wide Web. Note that human error can also have a significant effect on the security of your site (Liginlal, Sim, and Khansa, 2009).	26
System errors	25
Changes to source systems	20
Other	7
Source: Philip Russom, <i>Taking Data Quality to the Enterprise Through Data Governance</i> , The Data Warehousing Institute, 2005.	

The percentages do not add to 100 because they represent the relative number of survey respondents who reported the associated data quality problem as a significant source of data quality problems in their organization, not the percentage of all errors contributed by each source.

The surveys cover the same error sources except that the 2005 survey adds the category “Inconsistent definitions for common terms,” which tied with “Employee data entry” as the number one source of data quality problems in the organization.

Data Quality Problem Source	2002 Survey Score	2005 Survey Score	Percent Change
Data entry by employees	76	75	-1
Inconsistent definitions for common terms	0	75	0
Data migration or conversion problems	48	46	-2
Mixed expectations by users	46	40	-6
External data	34	38	4
Data entry by customers	25	26	1
System errors	26	25	-1
Changes to source systems	53	20	-33
Other	12	7	-5
Sources:			

Data Quality Problem Source	2002 Survey Score	2005 Survey Score	Percent Change
<ul style="list-style-type: none"> Wayne W. Eckerson, <i>Data Quality and the Bottom Line: Achieving Business Success Through a Commitment to High Quality Data</i>, Seattle, WA: The Data Warehousing Institute, 2002. Philip Russom, <i>Taking Data Quality to the Enterprise Through Data Governance</i>, Chatsworth, CA: The Data Warehousing Institute, 2005. 			

The category “inconsistent definitions for common terms” was not measured in the 2002 survey.

Note that over three years time, the percent change, whether positive or negative, is, with two possible exceptions, not significant. The exceptions are the relatively small 6% drop in “Mixed expectations by users” and the significantly large 33% drop in “Changes to source systems.”

It is interesting to note, however, that of the 79% of the organizations surveyed that have a data quality initiative in place, the team leading that initiative in the 2005 survey is most likely to be the data warehousing group, whereas in the 2002 survey, the data warehousing team was second to the IT department in terms of which was the more likely leader of the initiative. Unfortunately, a whopping 42% of those surveyed had no plans to institute a data governance initiative, while a mere 8% had such an initiative already in place. 33% had an initiative under consideration, while another 17% had such a plan in either its design or implementation phase.

Database Constraints and Enterprise Business Rules

Database constraints can easily validate data entries and enforce referential integrity, both of which are commonly reported sources of data errors. CHECK constraints can prevent a keypunch operator from successfully typing values into a table column that are outside the range of values permitted for that column by enterprise business rules, and referential integrity constraints can prevent child table rows from becoming orphaned as the result of a mistaken deletion of a parent table row or update to a parent table primary, or alternate, key.

Constraints are a physical implementation of the business rules under which an enterprise operates. Integrity constraints are a method of restricting database updates to a set of specified values or ranges of values. They ensure not only that bad data does not get into the database, but also that intertable relationships do not become corrupted by the improper deletion or updating of data from the existing database.

The basic types of integrity constraints are:

- Semantic
- Physical

Vantage also provides a referential constraint, which provides the Optimizer with a means for devising better query plans, but which does not enforce the referential integrity of the database (see the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Definitions

Term	Definition
Business rule	A component of the business model that defines specific conditional modes of activity. Business rules are expressed in natural language and are represented in a relational database by integrity constraints. They are the ultimate determining factor for proper database design.
Constraint	A predicate that must evaluate to TRUE if a database DELETE, INSERT, or UPDATE operation is to be permitted.
Integrity constraint	A component of the logical database model that formalizes business rules by specifying the boundary conditions and ranges permitted for various database parameters. Integrity constraints are one of the four components of an integrity rule and are expressed in the language of databases: tables, columns, rows, and so on.
Integrity rule	<p>A set of rules for ensuring the integrity of a database. Each integrity rule is composed of a:</p> <ul style="list-style-type: none"> • Name • Integrity constraint set • Checking time • Violation response <p>The checking time specifies the processing point at which the constraint is checked. In the ANSI/ISO SQL standard, a checking time has either of the following possible values:</p> <ul style="list-style-type: none"> • Immediate • Deferred <p>All Teradata constraints have an immediate checking time. Deferred constraint checking, which is never a good way to ensure integrity, is not permitted.</p> <p>The violation response specifies the action to be taken when an integrity constraint is violated. In the ANSI/ISO SQL standard, a violation response has either of the following possible values:</p> <ul style="list-style-type: none"> • Reject • Compensate <p>All Teradata constraints have a reject violation response. Compensate violation responses are not permitted.</p> <p>Integrity rules are specified by the SQL CREATE TABLE and ALTER TABLE statements.</p>

Semantic Data Integrity Constraints

Semantic integrity constraints enforce the logical meaning of the data and its relationships. Physical integrity constraints enforce the physical integrity of the data, for example ensuring that the bit sequence 00000001 is stored as 00000001 and not 00000010.

The emphasis of this section is on declarative semantic constraints: constraints that are part of the definition of the database itself. It is also possible to implement procedural support for integrity using database features such as triggers and stored procedures. The principal drawbacks to procedural enforcement of database integrity are performance, the added programming requirements and, critically, the increased opportunity for errors in enforcement introduced by these methods. If a declarative constraint alternative is available, it is almost always a better choice than a trigger (which can introduce serious sequential timing issues) or stored procedure.

The least favorable method for enforcing database integrity is through application programming. Besides the increased programming burden and its likely introduction of errors into integrity enforcement, application programming introduces the additional fault of application specificity. If one application enforces database integrity and another does not, or if two programs enforce integrity in different, perhaps contradictory ways, then a still greater chance of corrupting the database results. Worse still, application-based database integrity cannot affect ad hoc inserts, deletes, and updates to the database, and as a result place still further burdens on the DBA to find other mechanisms of preventing database corruption.

The point of relational databases is to be application-neutral, thus serving all applications equally well.

These constraints, in addition to the other constraints defined for the system during the process of normalization, such as functional and multivalued dependencies, have the added advantage of facilitating query and join optimization. The richer the constraint set specified for a database, the more opportunities there are to enhance query optimization.

The principal types of declarative constraints described are as follows:

- Column-level constraints
- Table-level constraints
- Database constraints

The recommendation for constraints is to specify them anywhere you can, being mindful of the performance debit their enforcement accrues. If you cannot trust your data, there is little point in maintaining the databases that contain it.

You can never declare semantic database constraints on columns defined with the XML, JSON, BLOB, or CLOB data types.

If performance issues make declarative constraints an unattractive option, then you should always implement integrity constraints by some other mechanism. The performance savings attained by implementing integrity constraints outside the database are often just transferred from the database to the application, negating any actual performance gains realized by not implementing the constraints declaratively.

The most important consideration must be that database integrity is consistently enforced.

Logical Integrity Constraints

The pursuit of theory is frequently perceived as being unnecessary in the day-to-day practice of administering relational database management systems, but a small dose of theory is essential to understanding the issues you encounter when designing and maintaining databases.

The extent to which practical relational database management can be formalized using the principles of set theory and formal logic is seldom appreciated. The correspondence between relational set theory and relational database theory is not always direct. While it is true that relational database theory is based on the relations of set theory, it necessarily introduces the additional constructs listed below.

- Database relations are typed, while set theory relations are not. Note that data types, or domains, are also a form of constraint.
- Database relations are not ordered left-to-right, while set theory relations are.

- Dependency theory is critical to database relations, while no corresponding theory exists for set theory relations.
- Relation variables are critical to database relations, while no corresponding theory exists for set theory relations. As a result, the concept of integrity as it is understood for relational database management is foreign to set theory relations.

Definitions of Terms

Before proceeding further, some definitions for terms used in predicate logic and the predicate calculus are in order.

Term	Definition
Assertion	See “Proposition”.
Existential quantifier	The symbolic quantifier \exists of predicate logic, signifying the logically equivalent English language phrases “for some,” “for any,” and “there exists.”
Identity predicate	The symbolic operator $=$ of predicate logic, signifying the logically equivalent English language phrases “is identical to” or “is equal to.”
Inference rules	The rule set of a formal system that determines the steps of reasoning that are valid for proving logical propositions.
Predicate	<p>A truth-valued function.</p> <p>The attributes of a relation (columns), as well as the relation heading (relation variable) itself, can be represented formally as logical predicates.</p> <p>This is true whether an explicit constraint is defined over the column or not, because when no explicit constraint is defined, the implicit constraint specified by the data type for the column specifies its minimum, essential, constraint. You cannot, for example, insert the character string ‘character string’ into a column typed as INTEGER without first converting the string into an integer value.</p> <p>This type of constraint is known as a domain constraint (see Domain Constraints).</p>
Predicate calculus	The set of inference rules by which propositions in predicate logic are proven.
Predicate logic	The study of statement validity using the truth-functional operators of the propositional calculus, the universal and existential quantifiers, and the identity predicate.
Proposition	<p>An assertion that can be proven unequivocally to be either true or false.</p> <p>In a relational table, or relation variable, all rows are assumed to be true propositions by default, because if they were false, they would have been prevented from entry in the database by the various integrity constraints, both implicit and explicit, defined on that database. Each proposition (tuple) in a relation is an instantiation of its relation variable predicate that evaluates to TRUE.</p> <p>This important property is sometimes called the Closed World Assumption (see The Closed World Assumption).</p>
Truth-valued function	A function that evaluates unequivocally to either TRUE or FALSE.
Universal quantifier	The symbolic quantifier \forall of predicate logic, signifying the English language phrase “for all.”

Relations, Relation Values, and Relation Variables

As the relational model has matured, careful analysis has continued to reveal facets of its definition that are obvious in hindsight, but had remained undiscovered and unstated until recently. An important example of such a revelation of the obvious is the relation variable, or *relvar*.

Note that relations, as defined in set theory, do not vary over time. A relation is a value, and values do not vary as a function of time. The number 3, for example, is always the number 3, and never 2 or 13 or 724.

Variables, on the other hand, take on different values as a function of time, so it is formally correct to state that the value of a relation variable changes as a function of time.

The distinction between a relation variable and its associated relation value is an important one because without it, the application of the axioms of traditional set theory to database relations is suspect at best. That set theory deals entirely with extensional sets: a set is determined entirely by its population. There is simply no notion of a set with changing population; each different population constitutes a different set."

In the context of business rules and their relationship with integrity constraints, the concept of the relvar is useful because it provides a formal, yet simple, framework in which to situate the idea of integrity constraints. Note that the relvar concept applies equally well to views (virtual relvars) and base tables (base relvars).

The first principle captured by the concept of the relvar is that the value of a relation is unrelated to its defining variable. The easiest way to explain this is by analogy. A variable in any programming language represents the possible values that can be taken for that variable, but in itself it is not those values: it is, instead, a place holder for them. The values represented by the program variable can take on any number of values, but the variable itself is always an abstraction.

Think of a relvar as the heading definition for the attributes of a relation: it is the ANDing of all the column headings, including their constraints, defined for the relation. In terms of logic, a relvar is the predicate for a relation. More specifically, a relvar is the internal predicate for a relation. The real world that relvar represents is its external predicate. Any lack of correspondence between the external and internal predicates for a relvar is the defining characteristic of the data quality issue (see the introduction to this section for more information) as well as the inspiration for the maxim that a database can only enforce consistency, not truth. Unless otherwise stated, the phrase *relvar predicate* refers to the internal relvar predicate in this document, while each individual instantiation of that relvar, a tuple (or row), is a proposition for that predicate that evaluates to TRUE.

The relation value, then, is the net value of all the data contained by the relation. Each time the relation is updated, its value changes, but its relation variable does not. In a very real sense, a relation becomes an entirely new relation when an update occurs, but its relation variable does not change.

For example, consider the following equation:

$$x + y = 7$$

If you set the variable $x=4$, then the value for variable y must be 3. If you then change the value of the variable x to 3, you have updated x , but you have not changed the value of 4 to 3, you have just changed the value of the variable x from 4 to 3. This holds for any other valid number over the domain of x . Another way to think of this is as follows: values are not time-varying. A 3 is always a 3 and never a 4. On the other hand, a variable

can represent any number of different values over the course of time. An old joke also illustrates this point by confusing variables with values: “Do you realize that $2 + 2 = 5$ for large values of 2”?

The integrity of this update relationship is maintained by the relvar predicate, which is defined by the logical ANDing of all the integrity constraints defined on the relation in question. The result of enforcing the relvar predicate is what Date and Darwen (1998, 2000) call The Golden Rule: no statement can leave any relvar with a value that violates its relvar predicate. This means that no update operation can ever cause any database constraint to evaluate to FALSE. In other words, the integrity of the database cannot be violated. The Golden Rule is one of the fundamental principles of relational database theory.

It follows that tuples (“rows” in everyday language: see [Definitions](#)) are instantiations of the relvar predicate that represent various propositions about the relation they constitute. When you think of a row as a proposition, it immediately follows that it must also be a true proposition because, by definition, false propositions (wrong, or corrupt data in everyday language) are not permitted in a database that enforces integrity constraints (see [The Closed World Assumption](#)). If they were, the database would not represent facts and the Golden Rule would be violated. In other words, if a given tuple is an element of a particular relvar, then that tuple, by default, satisfies its predicate by evaluating to TRUE.

Unfortunately, the representation of the real world by a database is only as good as its input, and any database will permit the insertion of factually wrong data as long as that data does not violate the constraints specified for the base table or view through which the table is updated (see [Sources of Data Quality Problems](#)). In other words, a database can only enforce consistency, not truth. All truths are consistent, but not all consistent things are true.

This is an important consideration: the database itself cannot know whether its information corresponds to real world facts (see [Sources of Data Quality Problems](#)). It can, however, constrain the data with which it is updated by ensuring that no business rules are violated, and it is these constraints that provide the mechanism for ensuring the success of the data integrity rules enforced by relational database management systems.

The Logic of Database Integrity

To ensure that no row in the database represents a logically false proposition, each table in the database must be capable of evaluating its own predicate, or relvar, for its truth value, and the only way it can enforce this integrity is to enforce the set of business rules defined by the enterprise it supports. In practice, this means that the predicate for any table is its sole criterion for update acceptability. From this, it follows that the formal meaning of any table is defined by the logical ANDing of its component column and table constraints. This logically ANDed constraint set is also the best approximation the database management system can make to any table predicate.

From this, it follows that the meaning of an entire database, its predicate, can be represented formally as the logical ORing of its table constraints logically ANDed with the set of all database constraints.

The Closed World Assumption

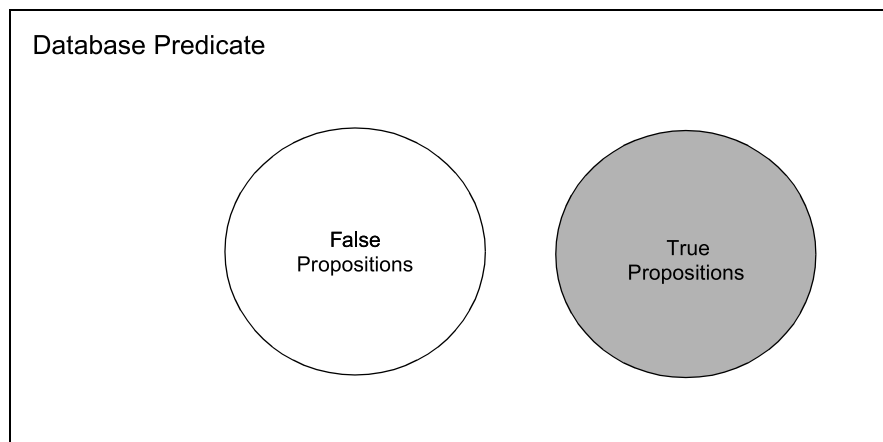
The Closed World Assumption, or CWA, asserts that if an otherwise valid tuple does not appear in the body of a relation, then the proposition representing that tuple must be false. In other words, the assumption is made that facts not known to be true in a relational database are false. Two other assumptions are: the

Unique Name Assumption, which states that any item in a database has a unique name and, further, that objects with different names are not the same; and the *Domain Closure Assumption*, which states that there are no objects other than those within the database. Two closely related rules are the *Completed Database Rule* and the *Negation As Finite Failure Rule*.

The CWA has become one of the fundamental principles of relational database theory.

In symbolic logic, such an assertion is called a completion axiom. It follows that a relation contains all of, and only those, tuples whose corresponding propositions are true. In other words, a relation body contains only those tuples that satisfy the completion axiom for its relvar. By generalization, the complete set of completion axioms for a given database defines its CWA, and the set of all tuples in a given database must also satisfy its CWA.

The following Venn diagram illustrates this point:



Note that, by the definition of the CWA, the sets of false and true propositions for a database do not intersect. See [The Closed World Assumption Revisited](#) for an application of the CWA to missing information in relational databases and the paradox that nulls present to the definition of a well-formed completion axiom.

How Relational Databases Are Built From Logical Propositions

The following example illustrates how a real database is built from logical propositions.

An Airline Reservation System

As a concrete example of the logical underpinnings of database constraints, consider an airline reservation. The following proposition can be stated about any reservation, and a particular instance of the proposition can be represented by a unique row in a table:

The reservation identified by reservation number (*reservation_number*), made for flight (*arrival_flight_number*), has a scheduled arrival date (*arrival_date*) and time (*arrival_time*) at gate (*arrival_gate_number*) and a scheduled departure flight of (*depart_flight_number*) with departure date (*depart_date*) and time (*depart_time*) from gate (*depart_gate_number*).

The variables enclosed within parentheses in this proposition are placeholders for the entire domain of values they represent. When you substitute actual values for these placeholders, you produce a proposition about a specific airline reservation. For example, suppose you have the following set of values:

(88079, 317, 10/19/2001, 13:59, 60, 1138, 10/25/2001, 05:40, 82)

When these values are substituted into the predicate framework, they produce the following logical proposition:

The reservation identified by reservation number 88079, made for flight 317, has a scheduled arrival date 10/19/2001 and time 13:59 at gate 60 and has a scheduled departure flight of 1138 with departure date 10/25/2001 and time 05:40 from gate 82.

This proposition, which represents an instantiation of the table predicate, might be represented in a "flight_reservations" table by the following row.

flight_reservations								
res_num	a_flight_num	a_date	a_time	a_gate	d_flight_num	d_date	d_time	d_gate
PK								
88079	317	10/19/2001	13:59	60	1138	10/25/2001	05:40	82

Several obvious constraints can be developed for the columns and table supporting this proposition.

Stated as simple sentences, these are the following.

- Column constraints:
 - Reservation numbers must be unique values drawn from a defined integer domain.
 - Arrival and departure flight numbers must be unique values drawn from a defined integer domain.
 - Arrival and departure dates must be unique values drawn from a defined date domain.
 - Arrival and departure times must be unique values drawn from a defined time domain.
 - Arrival and departure gate numbers must be unique values drawn from a defined integer domain.
- Table constraints, such as:
 - Departure dates must be greater than or equal to arrival dates.
 - Departure times must be greater than arrival times if they occur on the same date.
 - Arrival flight numbers and departure flight numbers cannot be equal if they occur on the same date.

Inclusion Compatibilities

Inclusion compatibilities are a generalization of referential constraints. As such, they provide the foundation upon which referential integrity is based. In common with functional compatibilities (see [Functional](#),

[Transitive, and Multivalued Compatibilities](#)), inclusion compatibilities represent one-to-many relationships (see [One-to-Many Relationships](#)); however, inclusion compatibilities typically represent relationships between relations (see [Database-Level Constraints](#)), while functional compatibilities always represent relationships between the primary key of a relation variable and its attributes.

Suppose you have the following table definitions:

Using the notation $R.A$, where R is the name of a relation variable and A is the name of one of its attributes, you can write the following inclusion compatibility:

$\text{supplier_parts.part_num} \rightarrow \text{parts.part_num}$

parts

PK				
part_num	part_name	color	weight	city

supplier_parts

PK		
FK	FK	
supp_num	part_num	quantity

This inclusion compatibility states that the set of values appearing in the attribute *part_num* of relation variable *supplier_parts* must be a subset of the values appearing in the attribute *part_num* of relation variable *parts*. Notice that this defines a simple foreign key-primary key relationship, though it is only necessary, in order to write a proper referential integrity relationship, that the right hand side (RHS) indicates any candidate key of the specified relation variable, not necessarily its primary key (see [Foreign Key Constraints](#) for more information about this).

The left hand side (LHS) and RHS of a compatibility relationship are not required to be a foreign key and a candidate key, respectively. This is merely required to write a correct inclusion compatibility expression of a referential integrity relationship.

Inference Axioms for Inclusion Compatibilities

Interference axioms for inclusion compatibilities are described in the following table:

Axiom	Formal Expression					
Reflexive rule	$A \rightarrow A$					
Projection and Permutation rule	IF	$AB \rightarrow CD$	THEN	$A \rightarrow C$	AND	$B \rightarrow D$
Transitivity rule	IF	$A \rightarrow B$	AND	$B \rightarrow C$	THEN	$A \rightarrow C$

Semantic Integrity Constraint Types

Business rules are expressed in relational databases by means of various types of integrity constraints. Four types of integrity constraint are supported:

- Domain
- Column
- Table
- Database

These four types of constraints are implemented in distinct ways, though the differences between column and table constraints are subtle.

Domain Constraints

Fundamentally, a domain is a data type. Data types act as simple constraints by not allowing you to, for example, insert a 20 character string into a field typed INTEGER.

The domain constraints for your databases are developed from the ATM Domains form. For more information, see [Constraints Form](#).

Column-Level Constraints

Column-level constraints define more elaborate integrity rules for columns than those defined by simple domain constraints. For example, a column constraint on a column typed as INTEGER might further specify that only values between 0 and 49999 or between 99995 and 99999 are permitted for that column.

A column constraint specifies a simple predicate that applies to one column only. For example, if some business rule states that employee numbers in the employee table must be between 10001 and 32001 inclusive. You could specify this rule in the CREATE TABLE statement used to define the emp_no column in the employee table as follows:

```
CONSTRAINT emp_no CHECK (emp_no >= 10001 AND emp_no <= 32001)
```

Notice that the only column referred to by this constraint is the employee number column, emp_no.

The column constraints for your databases are developed from the ATM Constraints form. For more information, see [Constraints Form](#).

You cannot declare column-level CHECK constraints on any column defined with the XML, JSON, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, ARRAY/VARRAY, Period, or Geospatial data types.

You can declare CHECK and UNIQUE constraints on row-level security constraint columns. When you define these constraints for a column, they apply to all rows in the table, not just to rows that are user-visible. You can also declare UNIQUE constraints on distinct and structured UDT columns as long as they are not based on XML, JSON, BLOB, or CLOB data.

You cannot declare database constraints other than NULL or NOT NULL for global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

You can also define column-level and object-level access constraints, or security privileges, on tables (for more information, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 and *Teradata Vantage™ - SQL Data Control Language*, B035-1149).

Row-Level Security Constraints

You can define row-level security constraints on tables. Row-level security is not a semantic constraint, but row-level security constraints and semantic constraints may interact. For details about row-level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Table-Level Constraints

Table-level constraints specify sets of relationships among values within a row. A table-level constraint can be either single-row or multirow, and applies to table columns rather than table rows.

For example, you might specify a table constraint on the `flight_reservations` table (see [An Airline Reservation System](#)) that requires the value for the scheduled arrival date, `a_date`, to be less than or equal to the value for the scheduled departure date, `d_date`. This is a single-row table constraint.

The SQL definition for this constraint looks like this:

```
CONSTRAINT arrive_date_check CHECK (a_date >= d_date)
```

Notice that this constraint refers to two columns, `a_date` and `d_date`; therefore, must be a table constraint.

You probably want to ensure that the value for `res_num` is unique and non-null, so you would define a uniqueness constraint on the table for reservation numbers. In this case, `res_num` is the primary key for the `flight_reservations` table, so the constraint is called a primary key constraint.

The SQL definition looks like this.

```
res_num INTEGER NOT NULL CONSTRAINT pKey PRIMARY KEY
```

Primary key constraints are a subset of uniqueness constraints: all primary keys are also unique, but all unique columns are not primary keys because a relation can only have one primary key. Because any unique column set is, by definition, also a candidate key, uniqueness constraints are sometimes referred to as key constraints. The primary key constraint on `res_num` is also a multirow table constraint because it enforces the rule that every row in the table has at least one unique value: its reservation number.

Suppose reservation number is not the primary key for the `flight_reservations` table, but your business rules require the values for `res_num` to be unique. To do this, you write a constraint that enforces uniqueness on the `res_num` column. The SQL definition looks like this:

```
res_num INTEGER NOT NULL CONSTRAINT ResNumUnique UNIQUE
```

The table constraints for your databases are developed from the ATM Constraints form. For more information, see [Constraints Form](#).

You cannot declare table-level CHECK constraints on any column defined with XML, BLOB, CLOB, UDT, Period, or JSON data types.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Database-Level Constraints

Database-level constraints specify some sort of functional determinant between the key and its dependent attributes (see [Functional, Transitive, and Multivalued Compatibilities](#)) as well as the functional determinants among two or more tables (see [Inclusion Compatibilities](#)).

The most commonly observed database-level constraint between tables is the primary key-foreign key relationship whose enforcement is referred to as maintaining referential integrity. This constraint specifies that you cannot delete a row having primary key value *x* from table *X* as long as any foreign key value in table *Y* has value *x* on the column set that defines the referential integrity relationship between those columns in tables *X* and *Y*. In other words, if you have foreign key value *x*, then you must also have primary key value *x* if referential integrity is defined for those keys and tables.

The specific table-level constraint syntax used to define the common PK-FK referential integrity constraint is `FOREIGN KEY (referencing_column_set) REFERENCES referenced_table_name (referenced_primary_key_column_name_set)`.

Though less frequently used, it is also possible to specify and enforce database-level constraints on non-PK-non-FK column relationships if the columns defining those relationships are alternate keys.

The specific constraint used to define an alternate key referential integrity constraint is a foreign key constraint with column-level syntax `REFERENCES table_name alt_key_name` and table-level syntax `FOREIGN KEY (referencing_column_set) REFERENCES referenced_table_name (referenced_alt_key_name)`, where *alt_key_name* refers to an alternate key column set in the parent table.

The database-level constraints for your databases are developed from the ATM Constraints form. For more information, see [Constraints Form](#).

You cannot declare database-level CHECK constraints on any column defined with XML, BLOB, CLOB, UDT, Period, or JSON data types.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Semantic Constraint Specifications

A name and a data type must be specified for each column defined for a table. Each specified column can be further defined with one or more attribute (see *Teradata Vantage™ - Data Types and Literals*, B035-1143) or constraint definitions.

There are several different specifications for constraints, some of which apply to multiple categories of constraints.

You cannot declare semantic database constraints on columns defined with XML, BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, XML-based UDT, ARRAY/VARRAY, Period, JSON, or Geospatial data types.

The following constraints are SQL column definition attributes that specify column-level integrity constraints:

- UNIQUE constraint definition on a single column.

UNIQUE constraints are implemented as USIs.

If a row-level security-protected table is defined with a UNIQUE constraint, enforcement of the UNIQUE constraint does not execute any row-level security policy defined for the table.

UNIQUE constraints are applicable to all rows in a row-level security-protected table, not just to user-visible rows.

You can specify UNIQUE constraints on columns having a UDT data type as long as the UDT is not based on XML, BLOB, or CLOB data.

You cannot define a UNIQUE constraint on a row-level security constraint column of a row-level security-protected table.

If you do not specify either an explicit PRIMARY INDEX or NO PRIMARY INDEX, Vantage converts any UNIQUE constraints you define to either a unique primary index or a unique secondary index, depending on whether a primary key is also defined for the table (see [Primary Index Defaults](#) for details).

IF PrimaryIndexDefault is set to D or P and a CREATE TABLE request specifies this constraint on a column set without also specifying either a PRIMARY INDEX or NO PRIMARY INDEX option ...	THEN Vantage converts the ...
PRIMARY INDEX	<p>column set defined as the primary key to the unique primary index for the table.</p> <p>Any additional column sets defined with UNIQUE constraints are redefined as unique secondary indexes.</p> <p>PRIMARY INDEX and UNIQUE constraints are implemented as unique secondary indexes. They are also explicitly redefined as unique secondary indexes in the SQL create text for the table definition.</p>
UNIQUE	<p>first column set defined with a UNIQUE constraint to the unique primary index for the table.</p> <p>Any other column sets defined with UNIQUE constraints are redefined as either unique secondary indexes.</p>

- CHECK constraint definition on a single column.

CHECK constraints are not implemented as indexes.

If a row-level security-protected table is defined with a CHECK constraint, enforcement of the constraint does not execute any security policy defined for the table.

CHECK constraints are applicable to all rows in a row-level security-protected table, not just to user-visible rows.

You cannot define a CHECK constraint on a row-level security constraint column of a row-level security-protected table.

- PRIMARY KEY constraint definition on a single column.

PRIMARY KEY constraints are implemented as USIs.

If a PRIMARY KEY constraint is defined where either or both the parent and child table are row-level security-protected, execution of the referential integrity constraint does not execute any security policy UDFs defined for the constraints on the table. Execution continues as if the tables were not row-level security-protected.

You can specify PRIMARY KEY constraints on columns having a UDT data type.

You cannot define a PRIMARY KEY constraint on a row-level security-protected column.

If you do not specify an explicit PRIMARY INDEX or NO PRIMARY INDEX option, Vantage converts any PRIMARY KEY constraint you define for a table to a unique primary index (see [Primary Index Defaults](#) for complete details).

- REFERENCES constraint definition on a single column.

REFERENCES constraints are not implemented as indexes.

If a REFERENCES constraint is defined where either or both the parent and child table are row-level security-protected, execution of the referential integrity constraint does not execute any security policy UDFs defined for the constraints on the table. Execution continues as if the tables were not row-level security-protected.

Temporal tables do not support foreign key REFERENCES constraints for standard or batch referential integrity. See *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details.

The following constraints are SQL table definition attributes that specify table-level and intertable integrity constraints:

- CHECK constraint definition on a composite column set.
- FOREIGN KEY ... REFERENCES constraint definition on a composite column set.
- PRIMARY KEY constraint definition on a composite column set.
- UNIQUE constraint definition on a composite column set.

You cannot specify constraints other than NULL or NOT NULL for global temporary trace tables (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

Performance Issues for Referential Integrity Constraints

The following set of topics describes some of the more important performance issues that are included in enforcing referential integrity.

For more information on referential integrity, see [The Referential Integrity Rule](#) and [Foreign Key Constraints](#).

Benefits of Referential Integrity

Benefit	Description
Maintains data consistency	Vantage enforces integrity relationships between tables based on the definition of a PK or a FK.
Maintains data integrity	When performing INSERT, UPDATE, and DELETE requests, Vantage maintains data integrity between referencing and referenced tables.
Increases development productivity	It is not necessary to code applications to enforce referential constraints because Vantage automatically enforces referential integrity.
Requires fewer programs to be written	Vantage prevents update activities from violating referential constraints. Vantage enforces referential integrity in all environments; you need no additional programs.

Overhead Costs of Referential Integrity

Overhead costs includes building the reference index subtables and inserting, updating, and deleting rows in the referencing and referenced tables. Overhead for inserting, updating, and deleting rows in the referencing table is similar to that of USI subtable row handling.

Vantage redistributes a row for each reference to the AMP containing the USI or reference index subtable entry. Processing differs after that, and most of the additional cost is in message handling.

When implementing tables with referential integrity, consider the following factors.

- Most importantly, the performance impact to update operations, which is frequently slowed when a referential integrity constraint must be enforced.
- INSERT performance slows for table because any referential integrity constraints defined for the table must be enforced.
- The cost of extra disk space for table maintenance resulting from referential integrity constraints.
- The cost of extra disk space for reference index subtables versus savings on program maintenance and increased data integrity.
- The cost of DML integrity validity checking in applications versus the cost of not checking.

Join Elimination and Referential Integrity

Join elimination is a process undertaken by the Optimizer to eliminate redundant joins based on information from referential integrity constraints.

The following conditions eliminate a join.

- A referential integrity relationship exists between the two tables.
- Request conditions are conjunctive, meaning they are ANDed rather than ORed.

This means that if any single condition in an ANDed set fails, the entire condition fails.

- The request does not contain reference columns from the PK table, other than the PK columns, including the SELECT, WHERE, GROUP BY, HAVING, ORDER BY columns.

- PK columns in the WHERE clause appear only in PK-FK joins.

IF...	THEN...
the preceding conditions are met	<ul style="list-style-type: none"> • the PK join is removed from the query. • all references to the PK columns in the query are mapped to the corresponding foreign key columns.
foreign key columns are nullable	Vantage adds a NOT NULL condition to the request.

Standard Referential Integrity and Batch Referential Integrity

In standard referential integrity, whether you are doing row-at-time updates or set-processing INSERT SELECT requests, each child row is separately matched to a row in the parent table, one row at a time. A separate SELECT request against the parent table is performed for each child row. Depending on your demographics, Vantage might select parent rows more than once.

With batch referential integrity, all of the rows within a single request, even if only one row is affected, are spooled, sorted, and their references checked in a single operation, as a join to the parent table. Depending on the number of rows in the INSERT ... SELECT request, batch referential integrity could be considerably faster, compared to checking each parent-child relationship individually.

For row-at-time updates, there is very little difference between standard referential integrity and batch referential integrity. But if you plan to load primarily using INSERT ... SELECT requests, batch referential integrity is recommended.

Referential Constraints

To maximize the usefulness of join elimination, you can specify Referential Constraints that Vantage does not enforce.

You can specify the REFERENCES WITH NO CHECK OPTION option to specify CREATE TABLE and ALTER TABLE statements with Referential Constraints, and the Optimizer can use the constraints without incurring the penalty of database-enforced referential integrity.

But when you use a REFERENCES WITH NO CHECK OPTION clause, Vantage does not enforce the Referential Constraints that you define. This means that a row having a non-null value for a referencing column can exist in a table even if an equal value does not exist in a referenced column. When you specify Referential Constraints, Vantage does not return error messages that would otherwise occur when RI constraints are violated.

If you specify a column name for a Referential Constraint, it need not refer to the single column PK of the referenced table or to a single column alternate key in the referenced table defined as UNIQUE, though best practice dictates that it should. The key acting as the PK in the referenced table need not be explicitly declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

The candidate key must always be unique even if it is not explicitly declared to be so, otherwise the referential integrity can produce incorrect results, and corrupt your databases if you do not take care to ensure that data integrity is maintained.

Specifying Referential Constraints relies heavily on your knowledge of your data. If the data does not actually satisfy the Referential Constraints that you provide, then requests can easily produce incorrect results.

The Optimizer can use the Referential Constraints without incurring the penalty of database-enforced referential integrity, but with the likelihood that Vantage can return corrupted result data.

Naming Constraints

Naming constraints is a good practice to follow because it permits a programmer to debug an embedded SQL or stored procedure application by fetching the name of a violated constraint from the SQLSTATE area. If constraints are not named, debugging is more difficult because it can be very difficult to determine which constraints belong to which system-defined names.

Row-level security constraints must be named. CHECK, UNIQUE, FOREIGN KEY, and PRIMARY KEY constraint specifications should be named.

Constraint names must conform to the rules for Vantage object names and be unique among all other constraint, primary index, and secondary index names specified on the table.

The characters used to name a constraint can be any of the following:

- Uppercase and lowercase alphabetic characters
- Integers
- Any of the following special characters.
 - -
 - #
 - \$

Vantage does not assign system-generated names to unnamed constraints.

CHECK Constraints

CHECK constraints are the most general type of SQL constraint specifications. Depending on its position in the CREATE or ALTER TABLE SQL text, a CHECK constraint can apply either to an individual column or to multiple columns.

Vantage derives a table-level index partitioning CHECK constraint from the partitioning expression for a PPI table. The text for this derived constraint cannot exceed 16,000 characters; otherwise, the system aborts the request and returns an error to the requestor. See [Partitioning CHECK Constraints](#) and [Partitioning CHECK Constraints for Single-Level Partitioning](#) for more information about this.

The following rules apply to all CHECK constraints.

- You can define CHECK constraints at column-level or at table-level.
- The specified predicate for a CHECK constraint must be a simple boolean search condition.

Subqueries, aggregate expressions, and CASE expressions are not valid search conditions for CHECK constraint definitions.

- You cannot specify CHECK constraints at any level for volatile tables or global temporary trace tables.
- Be aware that a combination of table-level, column-level, and WITH CHECK OPTION on view constraints can create a constraint expression that is too large to be parsed for INSERT and UPDATE requests.
- Vantage tests CHECK constraints for character columns using the current session collation.

As a result, a CHECK constraint might be met for one session collation, but violated for another, even though the identical data is inserted or updated.

The following is an example of the potential importance of this. A CHECK constraint is checked on insert and update of a base table character column, and might affect whether a sparse join index defined with that character column gets updated or not for different session character collations, in which case different request results might occur if the index is used in a query plan compared to the case where there is no sparse join index to use.

- Vantage considers unnamed CHECK constraints specified with identical text and case to be duplicates, and returns an error when you submit them as part of a CREATE TABLE or ALTER TABLE request.

For example, the following CREATE TABLE request is valid because the case of f1 and the case of F1 are different.

```
CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (F1>0));
```

The following CREATE TABLE request, however, is not valid because the case of the two unnamed f1 constraints is identical. This request aborts and returns an error to the requestor.

```
CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (f1>0));
```

- The principal difference between defining a CHECK constraint at column-level or at table-level is that column-level constraints cannot reference other columns in their table, while table-level constraints, by definition, must reference other columns in their table.
- If a row-level security-protected table is defined with one or more CHECK constraints, the enforcement of those constraints does not execute any UDF security policies that are defined for the table. The enforcement of the CHECK constraint applies to the entire table. This means that CHECK constraints apply to all rows in a row-level security-protected table, not just to the rows that are user-visible.

The following rules apply only to column-level CHECK constraints.

- You can specify multiple column-level CHECK constraints on a single column.

If you define more than one unnamed distinct CHECK constraint for a column, Vantage combines them into a single column-level constraint.

However, Vantage handles each named column-level CHECK constraint separately, as if it were a table-level named CHECK constraint.

- A column-level CHECK constraint cannot reference any other columns in its table.

The following rules apply only to table-level CHECK constraints.

- A table-level constraint usually references at least two columns from its table.
- Table-level CHECK constraint predicates cannot reference columns from other tables.
- You can define a maximum of 100 table-level constraints for a table at one time.

Foreign Key Constraints

Foreign key constraints permit you to specify referential primary key-foreign key relationships between a unique column set in the current base table and an alternate key column set in a different base table. The FOREIGN KEY keywords are required for table-level foreign key definitions but cannot be used for column-level foreign key definitions. If you specify FOREIGN KEY, then you must also specify a REFERENCES clause. Vantage uses referential integrity constraints to enforce referential integrity (see [The Referential Integrity Rule](#) and [Inclusion Compatibilities](#)) and to optimize SQL requests (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142). The overhead of enforcing standard referential integrity constraints is summarized in [Referential Integrity Constraint Checks](#) and [Overhead Costs of Standard Referential Integrity](#).

You can also use a foreign key definition to specify any of the following referential constraint types.

- Standard Referential Integrity
- Batch Referential Integrity
- Referential Constraints
- Temporal Relationship Constraints

Temporal tables can only be defined for so-called “soft” referential integrity relationships, or Referential Constraints, and for Temporal Relationship Constraints, and do not permit foreign key constraints for Standard or Batch Referential Integrity. See *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182 or for details.

When you specify the REFERENCES WITH NO CHECK OPTION phrase, Vantage does not enforce the defined referential integrity constraint. This implies the following things about child table rows:

- A row having a value, meaning that the implication is false if the referencing column is null, for a referencing column can exist in a table even when no equivalent parent table value exists.
- A row can, in some circumstances, match multiple rows in its parent table when the referenced and referencing column values are compared.

This can happen because the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See the list of rules for Referential Constraints later in this topic for details.

The various types have different applications as described in the following table.

Referential Constraint Type	Application
Referential Integrity Constraint	<ul style="list-style-type: none"> • Tests each individual inserted, deleted, or updated row for referential integrity. • If insertion, deletion, or update of a row would violate referential integrity, then AMP software rejects the operation and returns an error message.

Referential Constraint Type	Application
(see CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144)	<ul style="list-style-type: none"> Permits special optimization of certain queries.
Batch Referential Integrity Constraint (see CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144)	<ul style="list-style-type: none"> Tests an entire insert, delete, or update batch operation for referential integrity. If insertion, deletion, or update of any row in the batch violates referential integrity, then parsing engine software rolls back the entire batch and returns an abort message. Permits special optimization of certain queries.
Referential Constraint (see CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144)	<ul style="list-style-type: none"> Does not test for referential integrity. Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism. Permits special optimization of certain queries.
Temporal Relationship Constraint (TRC) (see <i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182)	<ul style="list-style-type: none"> Does not test for referential integrity. Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism. Permits special optimization of certain queries. TRC relationships can only be defined at the table level. TRC relationships cannot be defined on self-referencing tables.

Referential Constraints do not enforce primary key-foreign key constraints between tables, so they avoid the overhead of RI enforcement by the system as practiced by standard and batch referential integrity constraints. Their only purpose is to provide the Optimizer with a means for devising better query plans. Referential Constraints should be used only for situations for which referential integrity is either not important or is enforced by other means, because its use implicitly instructs the system to trust the validity of all DML requests made against the affected columns and not to check the specified referential integrity relationships.

Important:

If referential integrity errors occur, data corruption can occur. Erroneous results can be returned if a DML request specifies a redundant RI join and the primary key-foreign key rows do not match.

For more information, see [Inclusion Compatibilities](#) and the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The table on the following pages summarizes the differences among the different referential constraints.

Referential Constraint Type	CREATE/ ALTER TABLE Definition Clause	Description	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement	Join Elimination Optimizations?	Derived Statistics Propagated Between Child and Parent Tables?	Pros	Cons
Standard Referential Integrity Constraint	REFERENCES	ANSI/ISO SQL: 2011 compliant. Integrity enforcement done using a Referential Index (see Sizing a Reference Index Subtable).	Yes	Row	Yes	Yes	<ul style="list-style-type: none"> ANSI/ISO compliant. Logs RI violations in an error table. Efficient for low volume updates. 	Not efficient for medium to large updates because it is enforced one row at a time.
Batch Referential Integrity Constraint	REFERENCES WITH CHECK OPTION	Teradata extension to ANSI/ISO SQL: 2011 standard. All or nothing for update operations. Enforced using the following methods: <ul style="list-style-type: none"> Joining new child table keys to parent table to ensure they exist. Joining deleted parent table keys to child table to ensure they do not exist. 	Yes	Implicit transaction	Yes	Yes	Efficient for medium to large updates because uses the Optimizer to determine best way to make the join.	<ul style="list-style-type: none"> Can be slower than regular RI for small updates. Does not log RI violations in an error table.
Referential Constraint ("soft referential constraint")	REFERENCES WITH NO CHECK OPTION	Teradata extension to ANSI/ISO SQL: 2011 standard. No RI enforcement by Vantage. You must ensure the integrity of the relationship yourself. Not necessary to define a UNIQUE constraint for the primary or alternate key column set.	No	None	Yes	Yes	No cost for enforcing RI or uniqueness.	<ul style="list-style-type: none"> Incorrect results or data corruption can occur if the RI constraint is not valid. Not an issue if you are certain about the integrity of the relationship.
Temporal Relationship Constraint ("TRC constraint")							For more information on temporal relationship constraints, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186 and <i>Teradata Vantage™ - Temporal Table Support</i> , B035-1182.	

When you specify the REFERENCES WITH NO CHECK OPTION phrase, Vantage does not enforce the defined referential integrity constraint. This implies the following things about child table rows.

- A row having a non-null value for a referencing column can exist in a table even when no equivalent parent table value exists.
- A row can, in some circumstances, match multiple rows in its parent table when the referenced and referencing column values are compared.

This can happen because the candidate key acting as the primary key for the referenced table in the constraint need not be explicitly declared to be unique. See the list of rules for Referential Constraints later in this topic for details.

Referential Constraints do not enforce primary key-foreign key constraints between tables, so they avoid the overhead of RI enforcement by the system as practiced by standard and batch referential integrity constraints. Their only purpose is to provide the Optimizer with a means for devising better query plans. Referential Constraints should be used only for situations for which referential integrity is either not important or is enforced by other means, because its use implicitly instructs the system to trust the validity of all DML requests made against the affected columns and not to check the specified referential integrity relationships.

Note:

If referential integrity errors occur, data corruption can occur. Erroneous results can be returned if a DML request specifies a redundant RI join and the primary key-foreign key rows do not match.

For more information see [Inclusion Compatibilities](#) and the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Rules for Both Column-Level and Table-Level Foreign Key Constraints

The following rules apply to both column- and table-level FOREIGN KEY ... REFERENCES constraints:

- Teradata does not support the following ANSI/ISO SQL:2011 referential action options for FOREIGN KEY ... REFERENCES constraints:
 - MATCH {FULL | PARTIAL | SIMPLE}
 - ON UPDATE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}
 - ON DELETE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}
- The specified *column_name* list must be identical to a set of columns in the referenced table that is defined as one of the following.
 - PRIMARY KEY
 - UNIQUE
 - A unique secondary index

This rule is not mandatory for Referential Constraints. See [Foreign Key Constraints](#) for details.

The specified *table_name* refers to the referenced table, which must be a user base data table, not a view.

- A maximum of 64 foreign keys can be defined for a table and a maximum of 64 referential constraints can be defined for a table.

Similarly, a maximum of 64 other tables can reference a single table. Therefore, there is a maximum of 128 Reference Indexes that can be stored in the table header per table.

The limit on Reference Indexes includes both references to and from the table and is derived from 64 references to other tables plus 64 references from other tables = 128 Reference Indexes.

However, only 64 Reference Indexes are stored per Reference Index subtable for a table, those that define the relationship between the table as a parent and its children.

Column-level CHECK constraints that reference alternate keys in other tables do not count against this limit.

- Each individual foreign key can be defined on a maximum of 64 columns.
- Note the following attributes of foreign key constraints:
 - They can be null.
 - They are rarely unique.

An example of when a foreign key would be unique is the case of a vertical partitioning of a logical table into multiple tables.

- Each column in the foreign key constraint must correspond with a column of the referenced table, and the same column name must not be specified more than once.
- The referencing column list should contain the same number of column names as the referenced column list. The i^{th} column of the referencing list corresponds to the i^{th} column identified in the referenced list. The data type of each referencing column must be the same as the data type of the corresponding referenced column.
- The user issuing the CREATE TABLE request that specifies a foreign key constraint must either have the REFERENCE privilege on the referenced table or on all specified columns of the referenced table.
- If REFERENCES is specified in a column_constraint, then table_name defines the referenced table. Note that table_name must be a base table, not a view.
- Referential constraints are not supported for global temporary, global temporary trace, or volatile tables.
- While it is possible to create a child table at a time that its parent table does not yet exist, a REFERENCES constraint that makes a forward reference to a table that has not yet been created cannot qualify the parent table name with a database name.

In other words, the forward-referenced parent table that has not yet been created must be assumed to be “contained” in the same database as its child table that is currently being created.

- You cannot define a foreign key constraint on a row-level security-protected column.

Rules for Column-Level Foreign Key Constraints Only

The following rules apply to column-level foreign key constraints only.

- You cannot specify the keywords FOREIGN KEY or specify a referencing column set in the definition of a column-level foreign key constraint.

The referencing column for a column-level foreign key constraint is the column on which the foreign key constraint is defined by default.

- You cannot define a multicolumn foreign key constraint using the column-level foreign key syntax.
- If you omit `column_name`, the referenced table must have a single-column primary key, and the specified foreign key column references that primary key column of the referenced table.
- If you specify `column_name`, it must refer to the single-column primary key of the referenced table or to a single-column alternate key in the referenced table defined as `UNIQUE`.

This rule does not apply for Referential Constraints. In such cases, the candidate key acting as the primary key in the referenced table need not be explicitly declared to be unique using the `PRIMARY KEY` or `UNIQUE` keywords or by declaring it to be a USI in the table definition.

However, the candidate key in the relationship actually must always be unique even if it is not explicitly declared to be so. This is true by definition. See [Identifying Candidate Primary Keys](#) and [PRIMARY KEY Constraints](#) for details.

The uniqueness rule for candidate keys is necessary because the Optimizer always assumes that candidate keys are unique when it generates its query plans, and if that assumption is not true, it is not only possible to produce incorrect results to queries, but to corrupt databases. Because of this, you must always ensure that all candidate key values are unique even if they are not explicitly declared to be so.

As is always true when you specify Referential Constraints, you must assume responsibility for ensuring that any candidate key in a referential integrity relationship is unique, just as you must assume responsibility for ensuring that the referential relationship it anchors holds true in all cases, because Vantage does not enforce either constraint.

Rules for Table-Level FOREIGN KEY ... REFERENCES Constraints Only

The following rules applies to table-level `FOREIGN KEY ... REFERENCES` constraints only.

- You must specify a complete `FOREIGN KEY (referencing_column_set) REFERENCES (referenced_table_name)` specification for the foreign key definition.
- If you do not specify the optional `referenced_column_set` in the foreign key definition, Vantage assumes that the columns in the referencing column set have the identical names as the implied columns in the referenced column set.
- If the referenced column set columns do not have the same names as their counterparts in the `referencing_column_set` list, you must specify their names using the `FOREIGN KEY (referencing_column_set) REFERENCES referenced_table_name (referenced_column_set)` syntax.
- The optional keywords `WITH CHECK OPTION` and `WITH NO CHECK OPTION` define a foreign key constraint as being either a batch referential integrity constraint or a Referential Constraint, respectively.

If you specify neither set of keywords, the foreign key constraint defines a traditional foreign key constraint by default.

- A maximum of 100 table-level constraints can be defined for any table.

Rules for FOREIGN KEY ... REFERENCES Referential Constraints Only

Other than their not actually enforcing referential integrity, most of the rules for Referential Constraints are identical to those documented by *Rules for Both Column-Level and Table-Level Foreign Key Constraints* and by *Rules for Table-Level FOREIGN KEY ... REFERENCES Constraints Only*.

The exceptions are documented by the following set of rules that apply specifically to the specification and use of Referential Constraints.

- You can specify standard RI, batch RI, and Referential Constraints in the same table, but not for the same column set.
- You can specify Referential Constraints for both of the following constraint types:
 - FOREIGN KEY (*FK_column_set*) REFERENCES (*parent_table_PK_column_set*)
 - (*NFK_column_set*) REFERENCES (*parent_table_AK_column_set*)

where NFK indicates non-foreign key and *parent_table_AK_column_set* indicates an alternate key in the parent table.
- Referential Constraint references count toward the maximum of 64 references permitted for a table referenced as a parent even though they are not enforced by the system.
- INSERT, DELETE, and UPDATE requests are not permitted against tables with unresolved, inconsistent, or non-valid Referential Constraints. This rule is identical to the rule enforced for standard and batch RI.
- The candidate key acting as the primary key in the referenced table in the constraint need not be explicitly declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

Referential Integrity Constraint Checks

Vantage performs referential integrity constraint checks whenever any of the following things occur:

- A referential integrity constraint is added to a populated table.
- A row is inserted or deleted.
- A parent or foreign key is modified.

The following table summarizes these actions.

Action Taken	Constraint Check Performed
INSERT into parent table	None.
INSERT into child table	Must have matching parent key value if the foreign key is not null.
DELETE from parent table	Abort the request if the deleted parent key is referenced by any foreign key.
DELETE from child table	None.
UPDATE parent table	Abort the request if the parent key is referenced by any foreign key.
UPDATE child table	New value must match the parent key when the foreign key is updated.

Overhead Costs of Standard Referential Integrity

By implementing standard referential integrity, you incur certain overhead costs that can have a negative effect on performance. The following table lists the various referential integrity overhead operations that affect performance.

Operation	Description
Insert, update, or delete rows in a referencing table.	Overhead is similar to that for USI subtable row handling. A redistributed spool for each reference is dispatched to the AMP containing the subtable entry. BYNET traffic incurs the majority of the cost of this operation.
Insert a row into a referencing table	A referential integrity check is made against the Reference Index subtable. <ul style="list-style-type: none"> • If the referenced column is in the Reference Index subtable, the count in the Reference Index subtable is incremented. • If the referenced column is not in the Reference Index subtable, Vantage first checks the Reference Index subtable to verify that the referenced column exists. If it does, an entry with a count of 1 is added to the Reference Index subtable.
Delete a row from a referencing table	A referential integrity check is made against the Reference Index subtable and its count for the referenced field is decremented. When the count decrements to 0, then the subtable entry for the Referenced field is deleted.
Update a referenced field in a referencing table	Overhead is similar to that for changing the value of a USI column. A referential integrity check is made against the Reference Index subtable. Both the inserting-a-row and deleting-a-row operations execute on the Reference Index subtable, decrementing the count of the old Referenced column value and incrementing the count of the new Reference column value.
Delete a row from a referenced table	The Reference Index subtable is checked to verify that the corresponding Referenced column does not exist. When nonexistence is confirmed, the row is deleted from the Referenced table. No BYNET traffic is involved because the Referenced column is the same value in the Referenced table and the Reference Index subtable.

PRIMARY KEY Constraints

PRIMARY KEY constraints specify the primary key column set in a table definition. Vantage uses primary keys to enforce both row uniqueness and referential integrity.

Whether a PRIMARY KEY constraint is treated as a column-level constraint or a table-level constraint depends on whether the primary key is simple or composite.

The following rules apply to PRIMARY KEY constraints.

- Only one primary key can be defined per table.
- The following table explains the column limits for column-level and table-level primary key constraints.

IF the PRIMARY KEY constraint is ...	THEN you must define it at this level ...
simple, or defined on a single column	column. You can define a simple PRIMARY KEY constraint at table-level, but there is no reason to do so.
composite, or defined on multiple columns	table. Defining a table-level PRIMARY KEY constraint is the only way you can create a multicolumn primary key.

- Defining a primary key for a table is never required, though it is recommended for documentation purposes as part of a policy of enforcing data integrity in those cases where the logical primary key is not chosen to be the unique primary index.
- A primary key can be defined on a maximum of 64 columns.
- A PRIMARY KEY constraint cannot be defined on the same column set as the set used to define the nonunique primary index for a table.

PRIMARY KEY constraints are treated as ...	When the primary key is defined on this many columns ...
column-level constraints	1
table-level constraints	> 1

A maximum of 100 table-level constraints can be defined for any table.

- A primary key constraint can be defined on the same columns as a unique secondary index or unique primary index. A primary key is implemented as follows:
 - If the table is defined explicitly with a primary index or unique primary index on different columns than the primary key columns or if the table is defined as NO PRIMARY INDEX or PRIMARY AMP INDEX, then the primary key is implemented as a unique secondary index.
 - If a nonpartitioned, nontemporal table does not have an explicit primary index, explicit primary AMP index, or explicit NoPI and no USI or UNIQUE constraint is specified on the same columns as the primary key, then the default is a unique primary index on the same columns as the primary key constraint.

Note that in physical database design, candidate keys, whether chosen to be a primary key or not, are always defined internally as either a UNIQUE NOT NULL secondary index or as a single-table join index.

- You cannot define a PRIMARY KEY on a row-level security-protected column.

UNIQUE Constraints

UNIQUE constraints specify that the column set they modify must contain unique values. Vantage implements UNIQUE constraints as either a unique primary index, a unique secondary index, or as a single-table join index.

The following rules apply to UNIQUE constraints:

- UNIQUE constraints should always be specified with a NOT NULL attribute specification.

Otherwise, it is possible for a single null to be inserted into a uniquely constrained column. The semantics of a unique null “value” are uncertain at best, and almost certainly violate the intent of the uniqueness constraint.

- UNIQUE constraints can be defined at column-level (simple) or at table-level (composite).

The following table explains the column limits for column-level and table-level primary key constraints.

IF the UNIQUE constraint is ...	THEN you must define it at this level ...
simple, or defined on a single column	column. You can define a simple UNIQUE constraint at table-level, but there is no reason to do so.
composite, or defined on multiple columns	table. Defining a table-level constraint is the only way you can create a multicolumn UNIQUE constraint.

- Column-level UNIQUE constraints refer only to the column on which they are specified.
- Table-level UNIQUE constraints can be defined on multiple columns by specifying a column name list.
- A table-level UNIQUE constraint can be defined on a maximum of 64 columns.
- A maximum of 100 table-level constraints can be defined for any table.
- You can define a UNIQUE constraint for a column-partitioned table.
- If a row-level security-protected table is defined with a UNIQUE constraint, enforcement of the constraint does not execute any security policy defined for the table. UNIQUE constraints are applicable to all rows in a row-level security-protected table, not just to user-visible rows.
- You cannot define a UNIQUE constraint on a row-level security constraint column of a row-level security-protected table.

Semantic Constraint Enforcement

The enforcement of a constraint depends on how the base table on which it is defined is accessed.

If the base table is accessed directly, then its column and table constraints are always enforced. Date (2001a) calls this “The Golden Rule,” which he defines as follows: No update operation must ever assign to any database a value that causes its database predicate to evaluate to false. This, of course, is a generalization of the Closed World Assumption (see [The Closed World Assumption](#) and [The Closed World Assumption Revisited](#)).

By this definition, the checking time must always be immediate for any update; that is, the constraint check is made at statement boundaries, not deferred for checking at transaction (COMMIT time) boundaries. If this were not the case, then inconsistent, or false, data could be entered into the database, even if only for a brief time and even if the inconsistency were, as it must be, private to the transaction in question. It would still be

possible for a query that followed this inconsistent update within the boundaries of an explicit transaction to report erroneous information. Vantage does not support deferred integrity checking.

If a base table is accessed by means of a view, then the enforcement of any WHERE clause constraints specified in the view definition depends on whether the view is also defined WITH CHECK OPTION or not (see [Semantic Integrity Constraints for Updatable Views](#)).

Views inherit the constraints of their underlying base tables; therefore, base table constraints cannot be violated by updating through a view. However, additional constraints defined by means of a WHERE clause specification in a view definition can be bypassed if the view is not also defined WITH CHECK OPTION. The result is not a violation of any constraints defined on underlying base tables, but the insertion of a row that cannot be seen from that view.

Updatable Cursors and Semantic Database Integrity

Positioned updates using updatable cursors can present semantic integrity problems unless strict locking protocols are observed. Cursor updates (as used here, the word *update* also includes deletes) must be executed within the same transaction as the SELECT statement that defined the cursor.

Because Preprocessor2 does not support large objects, you cannot use cursors of any kind on columns defined with the XML, BLOB, or CLOB data types.

Between-Transaction Integrity Issues

By default, updatable cursors use READ locks, which are implemented as row hash locks internally. READ locks are adequate for preserving database integrity because they prevent access to the database by users who attempt to either change the definitions of database structures or to update table data. As a result, stored data cannot be changed while an open cursor is also manipulating the same data in such a way that it updates the same fields of the same rows. This is sometimes referred to as repeatable read mode. Programmers can also upgrade the locks used by updatable cursors to WRITE or even EXCLUSIVE locks if they so desire.

The potentially most severe problems for database integrity regarding updatable cursors are presented by the use of ACCESS locks, because these locks permit rows to be updated by other users while an open cursor is manipulating them, thus providing greater concurrency. The most important fact to understand about the various locking options for positioned updates via updatable cursors is that there is no integrity risk with READ and more restrictive locks, but there is an almost certain integrity risk with ACCESS locks.

Within-Transaction Integrity Issues

Between-Transaction Integrity Issues above describes the integrity risks presented by concurrent updating of a table by different transactions. This topic describes the risks associated with cursor conflicts that occur within an individual transaction.

Cursor conflicts occur in the following situations:

- Two cursors are open on the same table at the same time. One attempts a positioned update, but the other has already made a positioned update on the same row, thereby modifying the table.

- While a cursor is open, a searched update is made on a row, and then the cursor attempts a positioned update on that row.
- While a cursor is open, a positioned update is made on a row through that cursor and then a searched update is attempted on the same row.

All three of these updates are dirty because the same row was updated twice without the transaction having been committed.

Note that the system permits all three of the dirty updates to occur and then returns a cursor conflict warning to the requesting program. Resolution of the integrity breach is left to the user.

Because of the multiple possible integrity risks associated with declaring and opening multiple cursors within the same transaction, you should code your applications so those risks cannot occur.

Semantic Integrity Constraints for Updatable Views

Besides defining constraints on your base tables, you can specify additional constraints on a user-by-user basis through updatable views by specifying constraints in a WHERE clause and enforcing them by specifying a WITH CHECK OPTION clause. Because views inherit the constraints defined for their underlying base tables as well as those defined for any intermediate views, the constraints they inherit from their underlying relations are called derived constraints.

Specifying Integrity Constraints in an Updatable View Definition

Not all views are updatable. You can tailor update constraints at the user level by specifying user-specific constraints in the WHERE clause of a view as long as you also specify a WITH CHECK OPTION clause.

WITH CHECK OPTION pertains only to updatable views. Views are typically created to restrict which base table columns and rows a user can access in a table. Base table column projection is specified by the columns named in the column_name list, while base table row restriction is specified by an optional WHERE clause.

The WITH CHECK OPTION clause is an integrity constraint option that restricts the rows in the table that can be affected by an INSERT or UPDATE statement to those defined by the WHERE clause. If you do not specify WITH CHECK OPTION, then the integrity constraints specified by the WHERE clause are not checked for updates. Derived constraints inherited from underlying relations are not affected by this circumvention and continue to be enforced. The problem is that the view that updated the row in a way that violates the WHERE clause cannot view the updated row, so it cannot be updated in the future through that view.

WHEN WITH CHECK OPTION is ...	THEN any insert or update made to the table through the view ...
specified	only inserts or updates rows that satisfy the WHERE clause.
not specified	ignores the WHERE clause used in defining the view.

The following rules apply to updatable views and the WITH CHECK OPTION.

- If WITH CHECK OPTION is specified, the view is updatable. Any insert or update to the table through the view is rejected if the WHERE clause predicate evaluates to false.
- If WITH CHECK OPTION is not specified in an updatable view, then any WHERE clause contained in the view definition is ignored for any insert or update action performed through the view. In other words, the specified integrity constraints in the view definition are ignored, so you should always specify a WITH CHECK OPTION clause to define constrained updatable views unless you have an extraordinary reason not to.
- You can define nested views that only reference a single base table, which might allow the views to be updatable. In this case, the specification of a WITH CHECK OPTION clause in the view definition permits the WHERE clause constraints for that view, as well as those defined for any underlying views, to be exercised in the constraint on INSERT or UPDATE. See [Updatable View Inheritance](#) for more information.

The following request creates a view of the employee table so that it provides access only to the names and job titles of the employees in department 300:

```
CREATE VIEW dept300 (Name, JobTitle) AS
  SELECT name, job_title
  FROM employee
  WHERE dept_no = 300
  WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in this example prevents you from using the Dept300 view to update a row in the Employee table for which DeptNo <> 300.

This example shows how using a view in an UPDATE, INSERT, or DELETE statement allows you to add, change, or remove data in the base table set on which the view is defined.

Consider the following staff_info view, which provides a personnel clerk with retrieval access to employee numbers, names, job titles, department numbers, sex, and dates of birth for all employees except vice presidents and managers:

```
CREATE VIEW staff_info (number, name, position,
  department, sex, dob) AS
  SELECT employee.empno, name, jobtitle, deptno, sex, dob
  FROM employee
  WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
  WITH CHECK OPTION ;
```

If the owner of staff_info has the insert privilege on the employee table, and if the clerk has the insert privilege on staff_info, then the clerk can use this view to add new rows to employee.

For example, this request inserts a row into the underlying employee table that contains the specified information:


```
INSERT INTO staff_info (number,name,position,department,sex,dob)
VALUES (10024, 'Crowell N', 'Secretary', 200, 'F', 'Jun 03
1960');
```

Note:

The constraint on staff_info illustrated by the following WHERE clause applies to any insert using this view that includes the WITH CHECK OPTION phrase.

```
...
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
...
```

Therefore, the preceding INSERT statement would fail if the position entered for employee Crowell was 'Vice Pres' or 'Manager.'

Note:

If this view were defined without the WITH CHECK OPTION, and a user had the UPDATE privilege on the table, that user could update a job title to 'Vice Pres' or 'Manager'. The user would be unable to access the changed row through the view.

The following statement changes the department number typed for Crowell in the preceding INSERT request from 200 to 300:

```
UPDATE staff_info
SET department = 300
WHERE number = 10024;
```

Performing the following DELETE request removes the row for employee Crowell from the staff_info table.

```
DELETE
FROM staff_info
WHERE number = 10024;
```

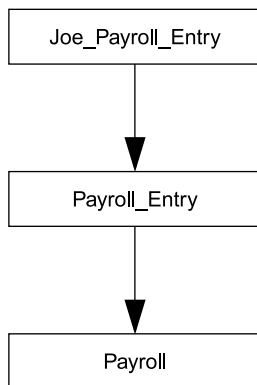
Views are a useful method for permitting selected users to have restricted access to base table data. However, as the preceding examples suggest, granting another user insert, update, and delete privileges on a view means relinquishing some control over your data. Carefully consider granting such privileges. The default is not to constrain updated or inserted values unless the view definition explicitly includes WITH CHECK OPTION.

Updatable View Inheritance

All base table integrity constraints are enforced without exception. Another way of stating this is to say that updatable views inherit the integrity constraints of their underlying base tables.

Because views can be nested, it is also true that updatable views inherit the integrity constraints of their underlying updatable views (when such relationships exist) as derived constraints in addition to inheriting the integrity constraints of their underlying base tables.

For example, suppose you have two nested views defined on the payroll table as follows.



When user Joe logs on, he is assigned the view `joe_payroll_entry` defined with the following `WITH CHECK OPTION` integrity constraint.

```
WHERE dept_no = 1350
WITH CHECK OPTION
```

Joe can only update values for department 1350.

View `joe_payroll_entry` is defined on top of view `payroll_entry`, which restricts the ability to update payroll for any employee who earns a base salary of 200,000 USD or greater. This constraint is specified by the following `WHERE` clause in the view definition:

```
WHERE base_salary < 200000
WITH CHECK OPTION
```

Because of this derived constraint, Joe can only update annual salaries less than 200,000 USD in department 1350.

View `payroll_entry` is defined on top of base table `payroll`, which restricts the ability to update payroll to only those employees who are US citizens or who have valid work visas. This constraint is specified by the following column-level `CHECK` constraint in the table definition:

```
visa_code CHARACTER(2)
CONSTRAINT ok2work CHECK (visa_code IN ('US','WV'))
```

Because of this additional derived constraint, Joe can only update salaries of US citizens or individuals with valid work visas who work in department 1350 and earn annual salaries less than 200,000 USD.

Summary of Fundamental Database Principles

This section and [The Normalization Process](#) has presented a number of fundamental principles of relational database management. The following table lists a summary of those rules:

Principle	Definition
Entity integrity rule	<p>The attributes of the primary key of a relation cannot be null.</p> <p>More accurately, this rule applies to any candidate key of a relation, not just to the candidate key chosen to be the primary key.</p> <p>Note that because this rule is explicitly intended to apply to base relations only and not virtual relations (views), it violates the Principle of Interchangeability.</p> <p>See Rules for Primary Keys.</p>
Referential integrity rule	<p>There cannot be any unmatched foreign key values.</p> <p>Restated more formally, assume a primary key value PK in relvar T_1 and a candidate key value FK in relvar T_2 that references it.</p> <p>The Referential Integrity Rule states that if FK references PK, then PK must exist.</p> <p>Note that the Referential Integrity Rule permits the attributes of FK to be wholly or partly null, which violates the Entity Integrity Rule because FK must reference a candidate key in T_1, and no attribute of a candidate key can be null.</p> <p>See The Referential Integrity Rule.</p>
Information principle	<p>A relational database contains nothing but relation variables. In other words, the information content of a relational database at any given instant is represented as explicit values (recall that nulls are not values) in attribute positions in tuples in relations.</p>
Closed world assumption	<p>A statement that is true is also known to be true. Conversely, what is not currently known to be true is false. See The Closed World Assumption and The Closed World Assumption Revisited.</p>
Principle of interchangeability	<p>No arbitrary or unnecessary distinctions shall be made between base relations (base tables) and virtual relations (views).</p>
Principles of normalization	<ul style="list-style-type: none"> • A relation variable that is not in 5NF should be decomposed into a set of 5NF projections. • The decomposition must be non-loss. • The decomposition must preserve dependencies. • Every projection on the non-5NF relvar must be required to reconstruct the original relvar from those projections. • The decomposition should stop as soon as all of its relation variables are in 5NF (or, situation permitting, 6NF).

Principle	Definition
Principle of orthogonal design	No two relations in a relational database should be defined in such a way that they can represent the same facts.
Assignment principle	After value x is assigned to variable V , the comparison $V=x$ must evaluate to TRUE.
Golden rule	No update operation can ever cause any database constraint to evaluate to FALSE. In other words, no statement can leave any relvar with a value that violates its relvar predicate. See Relations, Relation Values, and Relation Variables .
Principle of the identity of indiscernibles	Every entity has its own identity. In more formal terms, let E_1 and E_2 be any two entities. If there is no way to distinguish between E_1 and E_2 , then E_1 and E_2 are identical: they are one thing, not two.

Physical Database Integrity

Physical database integrity checking mechanisms usually detect data corruption caused by lost writes or bit, byte, and byte string errors. Most hardware devices protect against data corruption automatically by means of various error detection and correction algorithms. For example, bit- and byte-level corruption of disk I/O is usually detected, and often corrected, by error checking and correcting mechanisms at the level of the disk drive hardware, and if the corruption is detected but cannot be corrected, the pending I/O request fails.

Similarly, bit- and byte-level corruption of an I/O in transit might be detected by various parity or error checking and correcting mechanisms in memory and at each intermediate communication link in the path. Again, if the corruption is detected, but cannot be corrected, the pending I/O request fails.

CHECKSUM Integrity Checking and Physical Database Integrity

It is still possible for corrupted data to be written to the database, however. To minimize this problem, users can specify that checksums be performed on individual base tables. Checksums check the integrity of database disk I/O operations. A checksum is a numeric value computed from data. For a given set of data, the checksum value is always the same, if the data is unchanged.

Users can specify checksum for individual base tables in DDL using the ALTER TABLE, CREATE HASH INDEX, CREATE JOIN INDEX, and CREATE TABLE requests. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about using these statements.

Because calculating checksums requires system resources and may affect system performance, system-wide checksums are disabled by default on most platforms. Contact Teradata Services if you suspect disk corruption.

FALLBACK Protection and Physical Database Integrity

Fallback protection is another important data integrity mechanism. Fallback works by writing the same data to two different AMPs within a cluster. If the AMP that manages the primary copy of the data goes down, you can still access the fallback version from the other AMP.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

Keep in mind that if you specify fallback for a table, you double the amount of disk space required to store the same quantity of data. The amount of disk space required by a table is also doubled if you configure your system for RAID1 mirroring. This means that if you configure your disk for RAID1 mirroring and also specify fallback protection for a table, you actually quadruple the amount of disk space required to store the same quantity of data.

Also be aware that when a table is defined with fallback, it imposes a performance penalty for all DELETE, INSERT, and UPDATE operations on the table because each such operation must be executed twice in order to update both the primary table and its fallback table.

The system defaults to bringing Vantage up when AMPs are down on the assumption that any down AMPs can run in fallback mode. If your site does not use fallback for its critical tables, you probably want to keep Vantage down in this situation. To enable the logic to keep Vantage down, you can use the Maximum Fatal AMPs option from the SCREEN DBS command of the DBS Control utility (see *Teradata Vantage™ - Database Utilities*, B035-1102 for documentation of the SCREEN DBS command).

Vantage also provides a means for using fallback to deal with read errors caused by bad data blocks (see [About Reading or Repairing Data from Fallback](#)).

Disk I/O Integrity Checking

Not all problems with data integrity are the result of keypunch errors or semantic integrity violations. Problems originating in disk drive and disk array firmware can also corrupt user data, typically at the block or sector levels. Block- and sector-level errors are the most common origins of disk I/O corruption encountered in user data.

A major problem with handling this type of data corruption is that it generally is not detected until some time after it has occurred. As a result, queries against the corrupted data return semantically correct, but factually incorrect answer sets, and update or delete operations can either miss relevant rows or can change them in error. Once the system detects the corruption, the affected AMP is typically taken offline, various utilities such as ScanDisk and CheckTable are performed, and the data is either repaired or reloaded. Each of these actions either removes access to, or reduces the availability of, the data warehouse to its users until corrections have been made.

Levels of Disk I/O Integrity Checking

To protect against physical data corruption, Vantage permits you to select various levels of disk I/O integrity checking of your table, hash index, and join index data. Secondary index subtables assume the level of disk I/O integrity checking that is defined for their parent table or join index. These checks detect corruption of disk blocks (checksum sampling can also detect some forms of bit and byte corruption) using one of the following integrity methods, which are ranked in order of their ability to detect errors:

1. Full end-to-end checksums.

Detects lost writes and most bit, byte, and byte string errors.

2. Statistically sampled partial end-to-end checksums.

Detects lost writes and intermediate levels of bit, byte, and byte string errors.

3. No checksum integrity checking.

Detects some forms of lost writes using standard file system metadata verification.

Disk I/O Integrity Checking Detects and Logs Errors But Does Not Fix Them

This feature detects and logs disk I/O errors: it does not fix them. When the system detects data corruption, it removes the affected AMP from service and you must then take the appropriate measures to repair the corrupted data.

About Reading or Repairing Data from Fallback

When the file system detects a hardware read error, the AMP that owns the bad data block sends a message to the AMP that stores its fallback data. This message contains the tableID and rowID range of the unreadable data block. The fallback AMP reads its fallback rows in that range and sends them to the requesting AMP. There, the rows are reconstructed as a valid data block that can be used instead of the unreadable data block. In some cases, the system can repair the damage to the primary data dynamically. In other cases, attempts to modify rows in the bad data block fail. Instead, the system substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

To avoid the continued overhead of data block substitution when dynamic repair cannot be performed, rebuild the primary copy of table data manually from the fallback copy using the Table Rebuild utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102. You cannot use Table Rebuild to rebuild a hash index, join index, or USI, so if the corruption occurs in a hash index, join index, or USI subtable, you must drop the index and recreate it.

To detect all such read errors, the integrity checking level for the table or index should be set to ALL (see [Disk I/O Integrity Checking](#)).

Conditions That Support Reading or Repairing Data from Fallback

Reading or repairing data from fallback is limited to the following table, subtable, and data block types:

- Hashed tables.
- Primary hash index, join index, and USI subtable data blocks.

Reading or repairing data from fallback cannot be done for the following table types:

- Unhashed tables.
- BLOB, CLOB, or XML subtables.
- NUSI subtables, whether their parent table is defined with fallback or not.

Reading or repairing data from fallback can only be used when:

- All AMPs in the fallback cluster are up.
- Requests do not try to modify the data in the bad data block.

Designing for Missing Information

The efficacy and usefulness of the manner in which missing information is handled by SQL is in the eye of the beholder. Perhaps no other topic in relational database management generates as much controversy as does the dispute over the proper way to deal with missing information in relational databases. Independent of the semantic difficulties SQL presents when dealing with nulls, missing information also makes exploratory data analysis far more difficult than the identical data mining operation performed on data free of missing values.

The problems resulting from recording and manipulating missing information are concerned principally with the sometimes ambiguous, and often counterintuitive, meaning of nulls as reported by various types of database queries.

This section examines the various semantic ambiguities with nulls in order to make designers aware of the potential problems of interpretation they present. The principal thrust is to make you aware of the inconsistencies in the way SQL handles missing information. Careful use of nulls can provide benefits that cannot otherwise be achieved: at the same time, their careless use can present serious problems not only for the integrity of your databases, but also for the accuracy of the information you retrieve from them.

The recommendation for nulls is not to use them if you can avoid doing so. Constrain as many of your columns as possible to be NOT NULL. A carefully considered database design is often all that is required to avoid recording most nulls.

Semantics of SQL Nulls

For most applications, an SQL null means that the value of interest is not known. The only exceptions to this occur when nulls are used to represent the empty set. SQL does not support the empty set.

Types of Missing Values

The following list touches on most of the common uses of SQL nulls:

- Value is unknown
- Value is not applicable
- Value does not exist
- Value is not defined
- Value is not valid
- Value is not supplied
- Value is the empty set

The semantics, properties, and behavior of each of these null types are different, but SQL treats them identically, including all 14 of the ANSI/X3/SPARC “null manifestations” and their 8 submanifestations.

Inconsistencies in How SQL Treats Nulls

Recall that the defined semantics for SQL nulls is as the simple representation for missing information (see [Semantics of SQL Nulls](#)). A null is to be interpreted as missing information and nothing more. There are a number of areas in which the SQL treatment of nulls is inconsistent with this definition, and the purpose of this section is to summarize those inconsistencies.

The following summary of the inconsistencies in the treatment of nulls by SQL is intended to assist you with the interpretation of results that might otherwise be misleading:

- The empty set (notated symbolically as \emptyset), which is a well-defined *value* in set theory, evaluates as null in SQL. The empty set is an actual value in mathematics, not a place holder for missing information. The union of the empty set with any set **S** yields **S**. In other words, the empty set is the identity under union. In SQL, however, nulls are used to indicate both missing information (and many other things) as well as the empty set. To demonstrate that nulls and the empty set are not the same thing in SQL, try to submit the following request and see what happens.

```
SELECT column_name
FROM table_name
UNION
NULL;
```

Furthermore, in set theory, $\emptyset = \emptyset$, while in SQL, $\text{NULL} \neq \text{NULL}$.

For example, the evaluation of an empty character string in SQL returns null rather than the mathematically correct empty set. Similarly, aggregation over empty sets report null, which is meant to mean that the value is missing, but there are known results having real values for such operations on empty sets.

The problem with empty sets is also true for the outer join, where the extended columns in the join result are denoted by nulls, but are actually empty sets. As a result, nulls are used to represent both empty sets and missing information in the same report.

- Nulls have no value by definition, but sort as if they were all equal to one another.
- Similarly, when Vantage makes duplicate row checks, it treats nulls as if they were equal to one another.
- A null unique primary index is valid in Vantage. In this situation, the non-value NULL is treated as if it were a unique value, which it is not, because by definition, NULL is neither a value nor unique.

There can be only one null UPI per table, and if a table defined with a NUPI has very many null primary indexes, the distribution of those rows across the AMPs will be very skewed.

- With the exception of `COUNT(*)`, SQL aggregate operations ignore nulls, while SQL arithmetic operations do not.
- With the exception of `CASE`, `COALESCE`, and `NULLIF`, nulls are not valid predicate conditions in SQL expressions.
- With the exception of `CASE`, `COALESCE`, and `NULLIF`, SQL expressions cannot neither return nor operate on nulls.

- When a CASE, COALESCE, or NULLIF expression returns a null literal, it has INTEGER data type. All other nulls are untyped.
- Null is a relatively common European family name, so its use as a literal default for name columns can produce incorrect query results if there are instances of the family name Null in those columns.
- Structured UDTs can have null attributes, but the semantics of null attributes are different from the semantics of a null field in a column. For example, consider a structured UDT that is composed of a single attribute. If that attribute is set null, the field value in that column is *not* treated as a null with respect to the UDT column value.

You must explicitly place a null marker into the UDT column for the column “value” to be considered null.

The semantics of a null data type, whether partially or wholly null, are difficult to grasp. It can be said that a null attribute represents a missing type definition, but nulls are defined in ANSI/ISO SQL to represent missing values, not missing type definitions. The semantics of a UDT are what its designer defines them to be, of course, but from a logical perspective, it would seem that the best semantic definition of a null UDT would be undefined.

Using technology borrowed from object-oriented programming languages, it might be said that nulls superficially appear to be *overloaded* in SQL because multiple markers having different semantics are all subsumed under the same name: null. However, unlike the case for overloaded functions in object-oriented languages, it is not possible to discriminate among the various semantic possibilities a given null marker presents to a user or routine that must distinguish its intended semantics from among the myriad possible interpretations.

Bivalent and Higher-Valued Logics

This topic presents a brief overview of the relevant predicate logic underlying the database relational model and the place of nulls in that logic.

A two-valued logic (2VL) has 2 truth values: TRUE and FALSE. while a multivalued logic (MVL) has the truth values TRUE and FALSE plus an additional number, where the number of truth values depends on the individual logic.

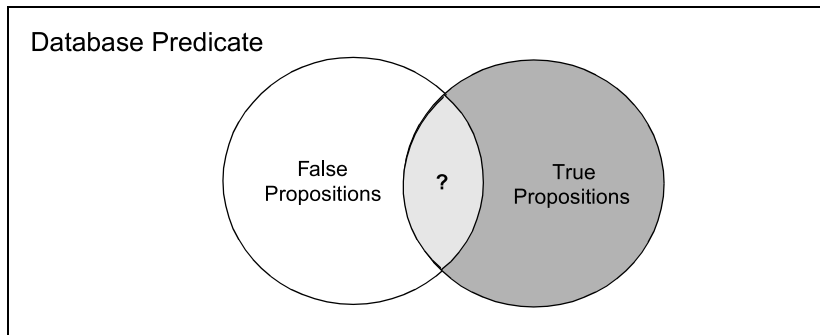
First Order Predicate Logic and Bivalent Logic

The classic first order predicate logic is based on bivalent logic (a bivalent, or Boolean, logic has exactly two truth values: TRUE and FALSE, as noted in the previous topic. Higher-valued logics have three or more truth values. For example, SQL uses a 3VL having the following three truth values: TRUE, FALSE, and UNKNOWN).

Upon close examination, it is apparent that the logic supporting SQL is actually a bivalent logic (2VL) with additional ad hoc support for nulls. Note that the inconsistent treatment of missing values in SQL is common to all vendors: it is not an implementation problem per se, but rather a problem with the way SQL itself is defined.

The Closed World Assumption Revisited

Recall that any tuple in a relation body is assumed to conform to the Closed World Assumption (see [The Closed World Assumption](#)). With the addition of nulls to the database relational model, the CWA can no longer be assumed to be true. As an example, consider the following Venn diagram.



In contrast with the Venn diagram in the topic *The Closed World Assumption*, there is overlap between the set of false propositions and the set of true propositions, represented by the darker shading. Better put, there is *potential* overlap between those sets. For example, suppose the *employee* table has an *emp_age* column that permits nulls. The company hires an employee who does not disclose her age in the application form. All other information for the employee is entered.

Because the age for this employee is missing, the tuple that represents her can neither be evaluated as true nor as false, so it falls into the ambiguous intersection of the Venn diagram.

At some point, it is discovered that this employee is 15 years old, which violates a constraint on the *emp_age* column that requires all employees to be at least 18 years old. The proposition for this employee now evaluates as FALSE, and the tuple that represents her is no longer valid in the database. Nevertheless, that tuple was treated as valid from the time its data was entered until the time the age of the employee was discovered to violate the constraint specified for the *emp_age* column, and that information was used to produce any number of reports which are, in retrospect, known to be non-valid.

Number of Logical Operators Supported for Bivalent and Trivalent Logics

In a bivalent logic, there are exactly four possible monadic, or single-operand, logical operators, as indicated in the following table. Note that the numberings have no meaning other than to distinguish the operators from one another.

This operator number ...	Performs these mappings of TRUE and FALSE values ...
1	$T \rightarrow T$ $F \rightarrow T$
2	$T \rightarrow F$ $F \rightarrow F$
3	$T \rightarrow F$ $F \rightarrow T$

This operator number ...	Performs these mappings of TRUE and FALSE values ...
4	$T \rightarrow T$ $F \rightarrow F$

By generalization, it is also true that there are exactly 16 possible dyadic, or two-operand, logical operators. The formulas that produce the numbers of monadic and dyadic operators are provided as follows, where n represents the valence of the logic in question. For a bivalent logic, its value is 2; for a trivalent logic, its value is 3.

Number of monadic operators = n^n

Number of dyadic operators = n^{n^2}

By performing the calculations for a trivalent logic, you find that 3VLs have 27 possible monadic operators and 19,683 possible dyadic operators. These figures, of course, are redundant: they represent the number of possible operators, not the number of useful operators.

SQL, like most programming languages, overdetermines its operators. For example, the function performed by the = operator in the following two equations is not identical internally, but the language uses the same symbol for the operation in both cases.

- 'value' = 'value'
- 2 = 2

The issue raised here is whether SQL supports a sufficient number of trivalent logical operators to fully support a 3VL, and the answer to that question is negative.

Nevertheless, they indicate that SQL does not support anywhere near the number of logical operators required to support a coherent 3VL.

SQL Support for a Consistent Trivalent Logic

The treatment of nulls by SQL is not only inconsistent, but it is also sometimes incorrect from a real world perspective.

The purpose of this description of the problems inherent in SQL missing values is to help you to understand what the most important inconsistencies are and how you can avoid some of them through careful database design.

Alternatives To Nulls for Representing Missing Information

Unlike every other feature of the database relational model, the treatment of missing information is not based on a formal foundation. As a result, even though the usual explanations of nulls often appear to make common sense, any detailed investigation reveals a number of potentially serious problems with SQL nulls.

The following sections describe some proposed alternative solutions to the missing value problem.

Systematic Use of Default Values

Because SQL has a built-in, nonprocedural, method to handle default values (see the default value control phrases `DEFAULT` and `WITH DEFAULT` in *Teradata Vantage™ - Data Types and Literals*, B035-1143), no special coding is required on the part of users. Note that if no defaults are specified for a column and it is not defined as `NOT NULL`, then SQL automatically inserts nulls to represent any information missing from an `INSERT` request.

Redesigning the Database to Eliminate the Need for Nulls

In the first order predicate calculus, propositions can have one of two values: they can either be true or they can be false. Recall that the rows of a relational table correspond to logical propositions that evaluate to `TRUE` (see [How Relational Databases Are Built From Logical Propositions](#)), because all false propositions are excluded from the database by means of various integrity constraints.

One can build a set of axioms deriving from such a set of true propositions in addition to a set of well-defined inference rules (referred to as a calculus in formal logic). Additional true propositions, formally known as theorems, can be derived from this calculus. These derived propositions correspond to valid relational queries if and only if the following statements about the original propositions are true:

- The initial set of rows represents only true propositions.
- All operations on a relation that contains tuples corresponding to these true propositions obey the formal inference rules of bivalent logic.

If either of these statements is false, then the correctness (truth of the derived propositions) of any query made against the data cannot be guaranteed. Because SQL nulls represent data values that cannot be asserted, it is not possible to know whether rows that contain nulls represent true propositions or not, so the tables that contain them are not true relations in the mathematical sense.

Noting that one of the most frequently provided justifications null supporters give for their use is that maintaining rows with missing values is often a necessary and useful thing to do in the real world, Pascal asks, why it would be considered useful to record what he calls “partial nothings” in the database if it is obviously nonsensical to maintain “full nothings”? Pascal then uses the following extreme example to support his case against nulls.

Taking the “Partial Nothing” Argument To Its Logical Extreme

Consider the following classic case study: an employee table that records the salary of each employee currently working for the enterprise. It is not uncommon for salaries to be unknown at a given point, so an employee row might contain a null to hold the place of the true salary of that employee. At the time the row is inserted into the table, the salary value is missing information.

At a given instant, this employee table might look like this:

employee

emp_num PK	emp_name	dept_num	hire_date	salary
214	Smith	32	09-12-1989	56150
447	Lau	15	05-30-1993	?
103	Hossein	09	09-13-1984	29775
500	Nakamura	11	06-09-1997	84932
713	Schroeder	24	10-29-2001	?

New hire Schroeder has not yet been assigned a salary and Lau has just been promoted, but her new salary has not yet been determined.

The following “partial nothing” table lists only the primary key values known for each row.

employee

emp_num PK	emp_name	dept_num	hire_date	salary
214	?	?	?	?
447	?	?	?	?
103	?	?	?	?
500	?	?	?	?
713	?	?	?	?

The problem with the “partial nothing” table is that its semantics are perceived as something other than what they truly are. The intended meaning of the nulls in the *salary* column derives from the true proposition that all employees receive a salary. The formal semantics of the table, however, assert only that a salary amount exists for each employee.

As a result, the table actually represents a mix of two different categories of assertion:

- Those rows with nulls that apply to all employees (“a salary amount exists for all employees”).
- Those rows that describe only those employees whose salaries are known (“all employees receive a salary”).

Another way of saying this is that multiple, inconsistent, informal predicates are being mapped (incorrectly!) into a single formal predicate. In this particular case, some rows (those containing nulls as place holders for the salary value) apply to all employees, while others (those that contain values for employee salaries) apply to a subset of that population. No single predicate can possibly apply to both situations.

Solving the Problem By Taking Simple Projections

The particular problem with nulls in this table is readily resolved with a minor change in the logical design of the database. Because the attributes in this *employee* table are a combination of those that apply to all tuples and those that apply only to some tuples, the relationship is similar to that of a simple entity supertype and subtype, the difference being that the attribute in question, *salary*, is not unique: all employees receive a salary.

The following tables resolve the problem of multiple simultaneous semantics for the original *employee* table by taking projections from the original *employee* table:

employee

emp_num	emp_name	dept_num	hire_date
PK			
214	Smith	32	09-12-1989
447	Lau	15	05-30-1993
103	Hossein	09	09-13-1984
500	Nakamura	11	06-09-1997
713	Schroeder	24	10-29-2001

employee_salary

emp_num	salary
PK	
214	56150
103	29775
500	84932

Manipulating Nulls With SQL

As described in [Types of Missing Values](#), SQL nulls are often mistakenly interpreted in ways that extend significantly beyond their single valid meaning, which is simply unknown (no value is present).

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements effect.

See [NULL Literals](#) for more details about manipulating nulls with SQL.

Logical and Arithmetic Operations on Nulls

Nulls are not valid as predicate conditions in SQL other than for the unique example of CASE expressions (see [Nulls and CASE Expressions](#)).

You cannot solve for the value of a null because, by definition, it has no value. For example, the expression `value = NULL` has no meaning and therefore can never be true (however, see [Null Sorts as the Lowest Value in a Collation](#) for a counterexample). A query that specifies the predicate `WHERE value = NULL` is not valid because it can never be true or false. The meaning of the comparison it specifies is not only unknown, but unknowable.

If you want to search for fields that do or do not contain nulls, you must use the operators `IS NULL` or `IS NOT NULL` (see [Searching for Nulls Using a SELECT Request](#), [Excluding Nulls From Query Results](#) and [Searching for Nulls and Nonnulls In the Same Search Condition](#)).

The difference is that when you use a mathematical operator like `=`, you specify a comparison between values or value expressions, whereas when you use the `IS NULL` or `IS NOT NULL` operators, you specify an existence condition. Note that even though `IS NULL` and `IS NOT NULL` return truth values, their operands are not truth values; therefore they are not logical operators and do not count against the number of possible logical operators for a trivalent logic calculated in [Number of Logical Operators Supported for Bivalent and Trivalent Logics](#).

Nulls and Arithmetic Operators and Functions

If an operand of any *arithmetic* operator or function is null, then the result of the operation or function is usually null. The following table provides some illustrations and some exceptions:

WHEN the expression is ...	THEN the result is ...
<code>5 + NULL</code>	null
<code>LOG(NULL)</code>	null
<code>NULLIFZERO(NULL)</code>	null
<code>ZEROIFNULL(NULL)</code>	0
<code>COALESCE(NULL, 6)</code>	6

Nulls and Comparison Operators

If either operand of a *comparison* operator is null, then the result is unknown and an error is returned to the requestor. The following examples indicate this behavior.

WHEN the expression is ...	THEN the result is ...	AND this error message returns to the requestor ...
<code>5 = NULL</code>	Unknown	3731
<code>5 <> NULL</code>		

WHEN the expression is ...	THEN the result is ...	AND this error message returns to the requestor ...
NULL = NULL		
NULL <> NULL		
5 = NULL + 5		

Note that if the argument of the NOT operator is unknown, the result is also unknown. This evaluation translates to FALSE as a final boolean result.

Nulls and Aggregate Functions

With the important exception of COUNT(*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them, and is one of the major inconsistencies in the way SQL deals with nulls.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

```
SUM(A) + (SUM B) <> SUM (A+B)
```

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are both null in the same rows, because in those cases the entire row is not counted in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(*), which does include nulls in its result.

If this property of nulls presents a problem, you can perform either of the following workarounds, each of which produces the desired result of the aggregate computation $SUM(A)+SUM(B) = SUM(A+B)$.

- Define all NUMERIC columns as NOT NULL DEFAULT 0.
- Use the ZEROIFNULL function nested within the aggregate function to convert any nulls to zeros for the computation, for example

```
SUM(ZEROIFNULL(x) + ZEROIFNULL(y))
```

Nulls and DateTime and Interval Data

The general rule for managing nulls with DateTime and Interval data environments is that, for individual definitions, the rules are identical to those for handling numeric and character string values.

WHEN any component of this type of expression is null ...	THEN the result is ...
Value	null.
Conditional	FALSE.
CASE	as it would be for any other CASE expression.

DateTime or Interval values are defined to be either atomically null or atomically non-null. For example, you cannot have an interval YEAR TO MONTH value in which YEAR is null and MONTH is not.

Nulls and CASE Expressions

The ANSI/ISO SQL-2008 definitions for the CASE expression and its related expressions NULLIF and COALESCE specify that these expressions can return a null. Because of this, their behavior is an exception to the rules for all other predicates and expressions.

The rules for null usage in CASE, NULLIF, and COALESCE expressions are as follows.

- If no ELSE clause is specified in a CASE expression and no WHEN clause evaluates to TRUE, then NULL is returned by default.
- Nulls and expressions containing nulls are valid as CASE conditions. The following examples are valid.

```
SELECT CASE NULL
      WHEN 10
      THEN 'TEN'
      END;
SELECT CASE NULL + 1
      WHEN 10
      THEN 'TEN'
      END;
```

- Nulls and expressions containing nulls are not valid as WHEN clause conditions. The following examples are not valid.

```
SELECT CASE column_1
      WHEN NULL
      THEN 'NULL'
      END
FROM table_1;
SELECT CASE column_1
      WHEN NULL + 1
      THEN 'NULL'
      END
FROM table_1;
SELECT CASE
      WHEN column_1 = NULL
```

```

        THEN 'NULL'
      END
    FROM table_1;
  SELECT CASE
        WHEN column_1 = NULL + 1
        THEN 'NULL'
      END
    FROM table_1;

```

The following example is valid.

```

  SELECT CASE
        WHEN column_1 IS NULL
        THEN 'NULL'
      END
    FROM table_1;

```

- In contrast to the situation for WHEN clauses in a CASE expression, nulls and expressions containing nulls are valid in THEN and ELSE clauses. The following example is valid.

```

  SELECT CASE
        WHEN column_3 = 'NULL'
        THEN NULL
        ELSE column_3
      END
    FROM table_1;

```

NULLIF and COALESCE

The behavior of the CASE shorthand expressions NULLIF and COALESCE is the same as that for CASE with respect to nulls.

See NULLIF and COALESCE in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for further information.

NULL Literals

The keyword NULL is sometimes available as a special construct similar to, but not identical with, a literal. In this case, the keyword NULL represents the SQL null placeholder for a value logically in an SQL request, but is not the same marker that the system stores to indicate missing information.

Rules for Using NULL as a Literal

The literal NULL can be used in the following ways.

- As a CAST source operand, for example.

```
CAST (NULL AS  value)
```

- As a CASE result, for example.

```
CASE expression
  THEN NULL
END
```

or

```
CASE expression
  THEN expression
  ELSE NULL
END
```

- As an item specifying a null is to be placed in a column on INSERT or UPDATE.
- As a default column definition specification, for example.

```
DEFAULT NULL
```

- As an explicit SELECT item, for example.

```
SELECT NULL
```

This usage is a Teradata extension to the ANSI/ISO SQL-2008 standard because it does not specify a FROM clause.

- As an operand of a function, for example.

```
SELECT TYPE(NULL)
```

This usage is a Teradata extension to the ANSI/ISO SQL-2008 standard because it does not specify a FROM clause.

Data Type of NULL Literals

When you use NULL as an explicit SELECT item or as the operand of a function, its data type is INTEGER. For example, if you perform SELECT TYPE(NULL), then the data type of NULL is returned as INTEGER.

In all other cases, NULL has no data type because it has no value.

Hashing on Nulls

Even though nulls have no external value, they do have an internal value that can be processed by the Vantage hashing algorithm. If a NUSI for a table permits nulls, there is an increased probability of an uneven distribution of the rows for that table across the AMPs of a system because all rows having a null primary index hash to the same AMP.

If the number of rows having a null primary index is sufficiently large, then significant skew occurs, making efficient parallel processing difficult to achieve. This is not a problem for UPIs because there can be no more than one null UPI per table.

Although indexes are not a logical concept, and therefore are not part of the relational model, Teradata primary indexes are built from in-row values, so a null UPI provides a mechanism for what is effectively a potential duplicate row to be stored in the system as long as the column set remains null.

Null Sorts as the Lowest Value in a Collation

When rows are sorted using an ORDER BY clause, nulls sort as the lowest value.

If any row has a null in the column being grouped, then all rows having a null are placed into one group. In other words, though it is syntactically incorrect to formulate the predicate `NULL = NULL` in a DML request because nulls have no value and therefore cannot be equated, it is also true that when SQL sorts nulls, the implicit result is that `NULL = NULL`.

Searching for Nulls Using a SELECT Request

The IS NULL operator tests for the presence of nulls in a specified column. For example, to search for the names of all employees who have a null in the `dept_no` column, you could type the following request.

```
SELECT name
FROM employee
WHERE dept_no IS NULL;
```

This query produces the names of all employees having a null in the `dept_no` column. Because all employees contained in the `employee` table have been assigned to a department, no rows would be returned for this particular example.

Searching for Nulls and Nonnulls In the Same Search Condition

To search for nulls and non-nulls within the same predicate, the search condition for nulls must be specified separately from any other search conditions.

For example, to select the names of all employees with the job title of 'Vice Pres,' 'Manager,' or null, you could type the following SELECT statement:

```
SELECT name, job_title
FROM employee
WHERE job_title IN ('Manager' OR 'Vice Pres')
OR job_title IS NULL;
```

Excluding Nulls From Query Results

The IS NOT NULL operator excludes rows having a null in a specified column from the results of a query.

For example, to search for the names of all employees with non-nulls in the job_title column, you could type the following request.

```
SELECT name
FROM employee
WHERE job_title IS NOT NULL;
```

The result of this query is the names of all employees with a non-null in the job_title column. For this particular example, the names of all employees are returned because every employee has, by definition, been assigned a job title.

Nulls and the Outer Join

The outer join is a join that retains rows from one or both of the joined tables, depending on whether it is declared to be a left, right, or full outer join, respectively. This means that rows that do not match the rows produced by the inner join are extended in the outer join result by reporting nulls in the nonmatching columns. From the perspective of set theory, these nonmatching column values are not missing values, but the empty set, which is a real value. Because SQL represents the empty set as a null, the semantics of outer joins must be evaluated carefully whenever there are actual missing values in the inner join portion of the operation.

Ambiguities presented by nulls in the outer join also make query optimization more difficult, because certain key methods of query optimization such as transitive closure often cannot be achieved.

Semantics of Nulls in the Outer Join

The motivation behind the outer join is to preserve information that is otherwise lost in an inner join. There can be no doubt that this is an important issue, but the SQL solution to the problem presents some inconsistencies regarding the semantics of the nulls reported by an outer join. The following contrast of simple inner and outer natural joins on the same two tables shows that the respective joins are between the following supplier and supplier_parts tables:

supplier

supplier			
suppl_num	suppl_name	status	city
PK			
S2	Jones	10	Paris
S5	Adams	30	Athens

supplier_parts

supplier_parts		
suppl_num	part_num	quantity
PK		
S2	P1	300
S2	P2	400

Here is the result of the inner join of the supplier and supplier_parts tables.

inner join supplier and supplier_parts

suppl_num	suppl_name	status	city	part_num	quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400

This join outcome has no information about supplier S5, therefore, it is said to have lost that information.

Here is the result of the natural outer join of the supplier and supplier_parts tables:

outer join supplier and supplier_parts

suppl_num	suppl_name	status	city	part_num	quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S5	Adams	30	Athens	?	?

This join outcome retains information about supplier S5, therefore it is said to preserve that information. Note that the result reports nulls for the part_num and quantity columns for supplier S5.

Think back to the definition of null semantics that was presented in [Semantics of SQL Nulls](#). Recall that the unequivocal definition for an SQL null is that it represents an unknown value. In this case, however, the values for both part_num and quantity for supplier S5 are known, and those values are both the empty set. Because SQL does not support empty sets directly, it substitutes nulls in their place. This is also true for the result of the length determination of an empty character string (see [Inconsistencies in How SQL Treats Nulls](#)).

The important thing to remember with respect to the semantics of outer join nulls is that nulls represent both the empty set (when used to extend the inner join to preserve information that would otherwise be lost) and semantically correct SQL nulls (when used to represent unknown information in the inner join portion of the result). The determination of which null is which is left to the user.

Note that in set theory, the empty set is sometimes referred to as the *null set*. It is extremely important to understand that the word null in this context means something entirely different from its meaning in the SQL

language. The null set is simply a set that contains no members. It is, therefore, empty, which is why this set is also referred to as the empty set.

Also note that there is only one empty set in set theory, analogous to how there is only one value of 3 in the set of real numbers, only one value of 3 in the set of cardinal numbers, and so on. Therefore, it is referred to as *the empty set*, not "an empty set." Note, too, that studies of the properties of the empty set are sometimes referred to as *nullology*. Again, this term has absolutely nothing to do with nulls as they are defined by the SQL language.

Using Data Compression

This section describes several data compression options that you can use to reduce disk space usage and, in some cases, also improve I/O performance.

- Multivalue compression (MVC)
- Algorithmic compression (ALC)
- Row compression
- Row header compression
- Autocompression
- Hash index and join index row compression
- Block-level compression (BLC)

Compression Types Supported by Vantage

Compression reduces the physical size of stored information. The goal of compression is to represent information accurately using the fewest number of bits. Compression methods are either logical or physical. Physical data compression re-encodes information independently of its meaning, while logical data compression substitutes one set of data with another, more compact set.

Compression is used for the following reasons.

- To reduce storage costs
- To enhance system performance

Compression reduces storage costs by storing more logical data per unit of physical capacity. Compression produces smaller rows, resulting in more rows stored per data block and fewer data blocks.

Compression enhances system performance because there is less physical data to retrieve per row for queries. Also, because compressed data remains compressed while in memory, the FSG cache can hold more rows, reducing the size of disk I/O.

Most forms of compression are transparent to applications, ETL utilities, and queries. This can be less true of algorithmic compression, because a poorly performing decompression algorithm can have a negative effect on system performance, and in some cases a poorly written decompression algorithm can even corrupt data.

Experience with real world customer production databases with very large tables indicates that compression produces performance benefits for a table even when more than 100 of its columns have been compressed.

Vantage uses several types of compression.

FOR this database element...	Compression refers to...
column values	<p>the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header. It applies to:</p> <ul style="list-style-type: none"> • Multivalue compression See Multivalue Compression. • Algorithmic compression See Algorithmic Compression. <p>You cannot apply either multivalue compression or algorithmic compression to row-level security constraint columns.</p>
hash and join indexes	<p>a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.</p> <p>See Row Compression.</p>
data blocks	<p>the storage of primary table data, or join or hash index subtable data. Secondary Index (SI) subtable data cannot be compressed.</p> <p>See Block-Level Compression.</p>
partition containers	<p>the autocompression method set determined by Vantage to apply to a container of a column-partitioned table or join index when you have not specified the NO AUTO COMPRESS option at the time the object was created.</p> <p>See Autocompression for further information about autocompression for column-partitioned tables and join indexes.</p>

Row compression, multivalue compression, block-level compression, and autocompression are lossless methods, meaning that the original data can be reconstructed exactly from the compressed forms, while algorithmic compression can be either lossless or lossy, depending on the algorithm used.

There is a small initial cost, but even for queries made against small tables, compression is a net win if the chosen compression method reduces table size.

For compressed spool, if a column is copied to spool with no expressions applied against it, then the system copies just the compressed bits into the spool, saving both CPU and disk I/O size. Once in spool, compression works exactly as it does in a base table. There is a compress multivalue in the table header of the spool that stays in memory while the system is operating on the spool. When algorithmic compression is carried to spool, the compressed data is carried along with the compress bits.

The column attributes COMPRESS and NULL (see *Teradata Vantage™ - Data Types and Literals*, B035-1143) are useful for minimizing table storage space. You can use these attributes to selectively compress as many as 255 distinct, frequently repeated column values (not characters), to compress all nulls in a column, or to compress both.

The limit of 255 values is approximate because there is also a limit on the number of bytes or characters per column that can be multivalue compressed. These limits vary for different types of character data, as the following table explains.

FOR this type of data ...	THE maximum storage per column is approximately ...
<ul style="list-style-type: none"> • BYTE 	4,093 bytes
<ul style="list-style-type: none"> • KanjiSJIS • GRAPHIC • Latin • Unicode 	8,188 characters

Identifying Uncompressed, Single-Value Compressed, and MultiValue Compressed Tables

To determine whether your existing tables present further opportunities for compression, you must first identify which of your current tables are compressed.

This SELECT request identifies uncompressed, single-value compressed, and multivalue compressed tables in databases *aaaa*, *bbbb*, and *cccc* that are 15GB or greater in size, as specified by the phrase `HAVING Current_Perm > 15000000000`. You should customize this specification to the requirements of your site.

```

SELECT dbt.DATABASENAME, dbt.TABLENAME,
       MAX(CASE WHEN (compressvaluelist IS NOT NULL)
                THEN (CASE WHEN INDEX(compressvaluelist,',') > 0
                           THEN '3. MVC '
                           ELSE '2. SVC '
                           END)
                ELSE '1. NONE'
              END) AS Compress_Type,
       MIN(pds.Current_Perm) AS Current_Perm
FROM DBC.columns AS dbt, (SELECT t.DATABASENAME, t.TABLENAME,
                                SUM(ts.CurrentPerm) AS Current_Perm
                           FROM DBC.Tables AS t, DBC.TableSize AS ts
                           WHERE t.DATABASENAME = ts.DATABASENAME
                           AND   t.TABLENAME = ts.TABLENAME
                           AND   ts.TABLENAME <> 'ALL'
                           HAVING Current_Perm > 15000000000
                           GROUP BY 1,2) AS pds
WHERE dbt.DATABASENAME IN ('aaaa','bbbb','cccc')
AND   dbt.DATABASENAME = pds.DATABASENAME
AND   dbt.TABLENAME = pds.TABLENAME
-- HAVING Compress_Type = '1. NONE'
GROUP BY 1,2
ORDER BY 1,3,4 DESC, 2;

```

The GROUP BY and ORDER BY conditions in this request order the data within each database by table size and compression type.

Note that the result set indicates that the only forms of compression found in databases *aaaa*, *bbbb*, and *cccc* in tables 15GB or greater in size are identified as NONE, SVC, and MVC, where NONE indicates no compression, SVC indicates single-value compression, and MVC indicates multivalue compression.

NONE identifies where no values are compressed; SVC identifies where values should be changed to MVC if possible to take advantage of the space savings from compressing more values in the table columns; MVC identifies where at least some space savings are achieved through multivalue compression.

Having a condition of MVC does not mean that an optimal multivalue compression state has been achieved for a table, however. You should make a serious effort to analyze the data from tables having a Compress_Type of MVC to determine whether still more space savings can be realized by multivalue compressing additional values in additional columns.

DatabaseName -----	TableName -----	Compress_Type -----	Current_Perm -----
aaaa	table_a	NONE	47,898,088,960
aaaa	table_e	SVC	16,797,542,912
aaaa	table_i	MVC	19,040,798,720
aaaa	table_j	MVC	18,537,593,856
aaaa	table_k	MVC	18,333,077,504
aaaa	table_1	MVC	15,982,028,800
bbbb	table_1	NONE	20,157,521,408
bbbb	table_2	SVC	113,774,842,368
bbbb	table_3	SVC	29,932,489,728
bbbb	table_4	SVC	19,060,238,848
bbbb	table_5	MVC	240,413,969,920
bbbb	table_6	MVC	191,052,663,296
bbbb	table_7	MVC	106,940,743,680
bbbb	table_8	MVC	102,247,339,520
bbbb	table_9	MVC	77,317,453,824
cccc	table_o	NONE	66,612,497,920
cccc	table_p	NONE	24,427,379,712

cccc	table_q	NONE	19,321,673,216
cccc	table_t	SVC	16,848,895,488
cccc	table_x	MVC	52,900,937,728
cccc	table_y	MVC	21,144,027,648

This topic is based on the article *Workload Toolkit—Part 4—Compression Analysis* on [Teradata Community - Developer Zone](#) by David Roth of the Teradata Professional Services organization. To identify block-level compressed tables, see [Finding Tables and Databases with Compressed Data Blocks](#).

Multivalue Compression

Multivalue compression (MVC) compresses repeating values in a column when you specify the value in a compression list in the column definition.

When data in the column matches a value specified in the compression list, the database stores the value only once in the table header, regardless of how many times it occurs as a field value for the column. The database then stores a smaller substitute value, often as small as 2 bits, in each row where the value occurs.

MVC generally provides the best cost/benefit ratio compared to other methods. Because it requires minimal resources to uncompress the data during query processing, you can use MVC for hot (frequently used) data without compromising query/load performance. MVC is also considered the easiest to implement of all the compression methods.

MVC is a logical data compression form and is lossless.

Besides storage capacity and disk I/O size improvements, MVC has the following performance impacts:

- Improves table scan response times for most configurations and workloads
- Provides moderate to little CPU savings

Procedure

1. Use the COMPRESS phrase in a CREATE TABLE or ALTER TABLE statement to specify a list of frequently occurring values for the column that contains the values, for example:

```
CREATE TABLE Employee
  (EmpNo INT NOT NULL,
   EmpLoc CHAR(30) COMPRESS ('Seattle','Dallas','Chicago')
  ...
   DOB DATE COMPRESS (DATE '1972-02-29', DATE '1976-02-29')
  ...);
```

2. The system automatically compresses the specified values when data moves into the table, and uncompresses them when the containing rows are accessed.

Note:

The system automatically compresses NULLs when you specify COMPRESS.

Note:

Compressing many values in a table can cause the dictionary cache to overflow. If this happens increase your dictionary cache to the default value of 1 MB.

Note:

You cannot apply MVC to row-level security columns.

You can use MVC to compress columns with these data types:

- Any numeric data type
- BYTE
- VARBYTE
- CHARACTER
- VARCHAR
- DATE

To compress a DATE value, you must specify the value as a Date literal using the ANSI DATE format (DATE 'YYYY-MM-DD'). For example:

```
COMPRESS (DATE '2000-06-15')
```

- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE

To compress a TIME or TIMESTAMP value, you must specify the value as a TIME or TIMESTAMP literal. For example:

```
COMPRESS (TIME '15:30:00')
```

```
COMPRESS (TIMESTAMP '2006-11-23 15:30:23')
```

In addition, you can use COMPRESS (NULL) for columns with these data types:

- ARRAY
- Period
- Non-LOB distinct or structured UDT

Multivalue Compression Example

The following CREATE TABLE fragment specifies that all occurrences of 'cashier', 'manager', and 'programmer' for the jobtitle column as well as all nulls are to be compressed to zero space.

Nulls are compressed by default whether or not there is an argument to the COMPRESS attribute specified for a column.

This definition saves 30 bytes for each row whenever an employee has one of the following job titles:

- Null
- Cashier
- Manager
- Programmer

```
CREATE TABLE employee (
  employee_number INTEGER
  ...
  jobtitle          CHARACTER(30) COMPRESS ('cashier',
                                             'manager', 'programmer')
  ...
);
```

Guidelines for Using Multivalue Compression

- To change the values that are compressed or to change whether or not a column uses compression, use an ALTER TABLE request. For example, the following turns off the compression in the mycol_varchar column:

```
ALTER TABLE DB.mytable_vlc
ADD mycol_varchar NO COMPRESS;
```

- You cannot compress more than 255 distinct values for an individual column.
- You cannot create a table with more bytes compressed than there is room to store them in the table header.
- The maximum number of characters that can be listed in a COMPRESS clause is 8,188.
- You cannot compress values in columns that are any of the following:
 - Primary index columns
 - Identity columns
 - Derived table columns
 - Derived period columns
 - Row-level security constraint columns
 - Referenced primary key columns
 - Referencing foreign key columns for standard referential integrity relationships

You can compress values in referencing foreign key columns for Batch and Referential Constraint referential integrity relationships.

- You can compress columns that are a component of a secondary index, but MultiLoad operations on a table with a secondary index can take longer if the secondary index column set is compressed.

To avoid this problem, drop any compressed secondary indexes before starting the MultiLoad job and then recreate them afterward.

- You can compress columns that are components of a referential integrity relationship.
- You can assign multivalue compression to columns that contain the following types of data:
 - Nulls, including nulls for distinct and structured non-LOB and non-XML UDTs, ARRAY/VARRAY and Period UDT data types
 - Zeros
 - Blanks
 - Constants having any of the data types supported by multivalue compression.

Tradeoffs Between Multivalue Compression and Storage Requirements for Compressed Values

About Multivalue Compression and Net Capacity for Nulls and Values

While multivalue compression removes specified values from row storage, those values do not disappear: they must be stored somewhere. This statement applies only to values, not nulls. Null compression is handled by the presence bits in the row header and does not have an impact on the table header.

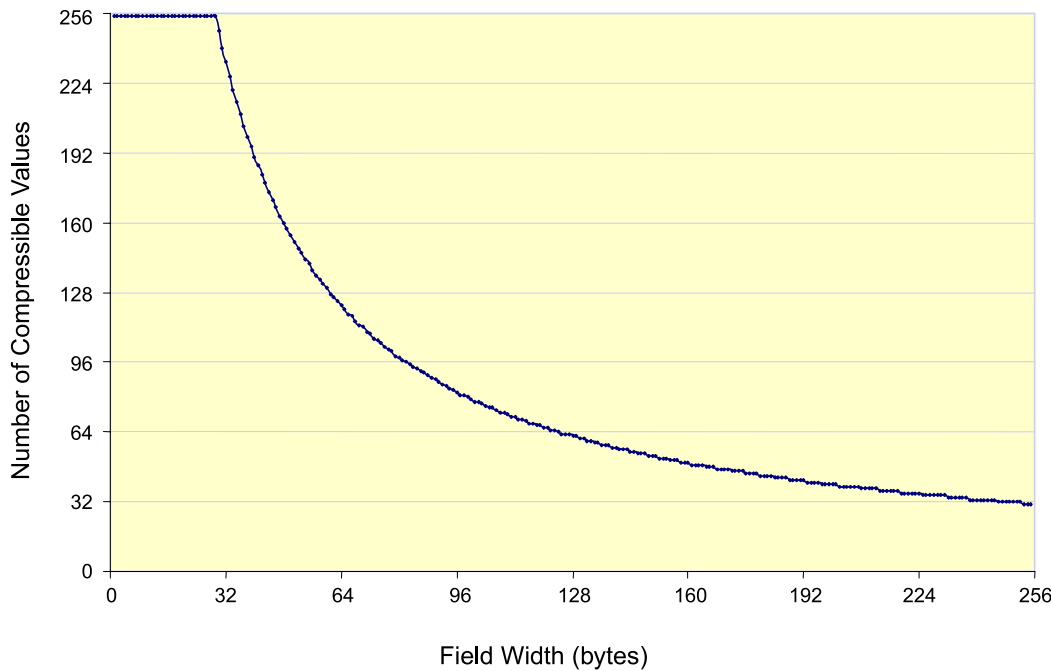
Storage of Compressed Values

The presence bits in the row header index into field 5 of the table header, where the compressed values are stored, once per column per AMP. This does not apply to algorithmically compressed data, which is stored in place within the row except for algorithmically compressed BLOB, BLOB-related UDT, CLOB, CLOB-related UDT, XML, XML-related UDT, or Geospatial data that is generally stored in subtables.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about algorithmic compression.

Because the size of the table header is limited to 1 MB, there is a limit to how many bytes can be compressed for a given column. If the number of bytes compressed exceeds the maximum row length, then the CREATE or ALTER TABLE statement used to create the new table is not valid and the DDL statement aborts. This is true even if the number of values specified for compression does not exceed the upper limit of 255.

The following graph plots the number of compressible values that can be specified for a column as a function of column width.



Not surprisingly, the plot clearly indicates that the wider the column, the fewer the number of values that can be compressed for the column. Particularly for wider columns, this means that in order to optimize compression, you must carefully analyze your tables to determine which values occur the most frequently and then limit compression to the top n values from that list.

Algorithmic Compression

Vantage software includes several standard compression algorithms, in the form of UDFs, which you can use to compress many types of data by table column. You can also create custom compression and decompression algorithms in UDF format.

When column values are mostly unique, algorithmic compression (ALC) may provide better compression results than MVC. When columns have repeated values, you can use ALC and MVC concurrently on the same column, but the system does not apply ALC to any value covered by MVC.

ALC generally functions best on cold (seldom used) data because of the amount of CPU required for decompress/recompress when compressed data is accessed, although some ALC algorithms require less CPU than others. ALC is considered to be the most difficult to implement of all the compression methods.

You can use algorithmic compression to compress table columns with the following data types:

- ARRAY
- BYTE
- VARBYTE
- BLOB
- CHARACTER

- VARCHAR
- CLOB
- JSON, with some restrictions listed below
- DATASET, with some restrictions listed below
- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE
- Period types
- Distinct UDTs, with some restrictions listed below
- System-defined UDTs, with some restrictions listed below

Procedure

1. Specify the name of an ALC compression UDF in the COMPRESS USING phrase for a column definition in a CREATE TABLE or ALTER TABLE statement, for example:

```
CREATE TABLE table_name
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE,
   Description VARCHAR(1000)
    COMPRESS USING TD_SYSFNLIB.compression_UDF_name
    DECOMPRESS USING TD_SYSFNLIB.decompression_UDF_name);
```

2. Specify the name of an ALC decompression UDF in the DECOMPRESS USING phrase for the column definition.

Usage Notes

- Specify only one set of compression/decompression UDFs for a particular column.
- The system automatically compresses nulls when you specify COMPRESS.

Note:

Teradata standard compression UDFs are located in the Teradata system function library, TD_SYSFNLIB.

For information about functional details, usage requirements, and restrictions on compression and decompression functions, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Restrictions

These restrictions apply:

- You cannot use ALC to compress columns that have a data type of structured UDT.

- The TD_LZ_COMPRESS and TD_LZ_DECOMPRESS system functions compress all large UDTs including UDT-based system types such as Geospatial, XML, and JSON. However, if you write your own compression functions, the following restrictions apply:
 - Custom compression functions cannot be used to compress UDT-based system types (except for ARRAY and Period types).
 - Custom compression functions cannot be used to compress distinct UDTs that are based on UDT-based system types (except for ARRAY and Period types).
- You cannot write your own compression functions to perform algorithmic compression on JSON type columns. However, Teradata provides the JSON_COMPRESS and JSON_DECOMPRESS functions that you can use to perform ALC on JSON type columns.
- You cannot write your own compression functions to perform algorithmic compression on DATASET type columns. However, Teradata provides the SNAPPY_COMPRESS and SNAPPY_DECOMPRESS functions that you can use to perform ALC on DATASET type columns.
- You cannot use ALC to compress temporal columns:
 - A column defined as SYSTEM_TIME, VALIDTIME, or TRANSACTIONTIME.
 - The DateTime columns that define the beginning and ending bounds of a temporal derived period column (SYSTEM_TIME, VALIDTIME, or TRANSACTIONTIME).

You can use ALC to compress Period data types in columns that are nontemporal; however, you cannot use ALC to compress derived period columns.

For more information about temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

- You cannot specify multivalued or algorithmic compression for a row-level security constraint column.

You can apply algorithmic compression to referential integrity columns.

Depending on the implementation, algorithmic compression can be either physical or logical, though most implementations use physical data compression. Algorithmic compression can be either lossy or lossless, depending on the algorithm used.

Using Custom ALC Algorithms

You can implement a custom compression algorithm as a scalar external UDF, and then specify the UDF in the COMPRESS USING phrase of a column definition for a CREATE TABLE or ALTER TABLE statement, for example:

```
CREATE TABLE table_name
  (ItemNo INTEGER,
   Type CHAR(10) UPPER CASE,
   Description VARCHAR(1000)
    COMPRESS USING compression_UDF_name
    DECOMPRESS USING decompression_UDF_name);
```

Note:

If you create custom ALC UDFs, test them thoroughly. If the compression or decompression algorithm fails, compressed data may be corrupted or may not be recoverable.

Related Information

For information on...	See...
specifying a UDF in the COMPRESS USING phrase of a CREATE TABLE or ALTER TABLE statement	<ul style="list-style-type: none"> • <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 • <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i>, B035-1184
rules for using compression/decompression algorithms in UDFs	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147

Topic	Reference
System-defined external UDFs for algorithmic compression and decompression	<i>Teradata Vantage™ - SQL Operators and User-Defined Functions</i> , B035-1210
How to code scalar UDFs to perform algorithmic compression on column data	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147
How to create the SQL definition for an algorithmic compression UDF	CREATE FUNCTION (External Form) information in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
How to specify those scalar UDFs in a table definition	ALTER TABLE and CREATE TABLE information in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
Guidelines for selecting an algorithmic compression functions	https://downloads.teradata.com/extensibility/articles/selecting-an-alc-compression-algorithm
Evaluating algorithmic compression UDFs	Download the test suite from https://downloads.teradata.com/download/extensibility/algorithmic-compression-test-package

Row Compression

Row compression is a form of logical data compression in which Vantage stores a repeating column value set only once, while any non-repeating column values that belong to that set are stored as logical segmental extensions of the base repeating set. Row compression is a lossless method.

Like multivalue compression, there is no decompression necessary to access row compressed data values. For an explanation of row compression, see the information about CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Note:

You can use row compression for hash and join indexes that are defined on row-level security-protected columns.

You control the row compression of join indexes with the syntax you use when you create an index (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144). The row compression of hash indexes is automatic and requires no special syntax.

Row Header Compression

The term *row header compression* applies only to column-partitioned tables and column-partitioned join indexes, and only when their data is stored in COLUMN format. Row header compression is a lossless method.

Vantage packs column partition values into a containers up to a system-determined limit. When that limit is reached, it begins packing column partition values into a new container.

The column partition values packed into a container must be in the same combined partition to be packed into a container. The row header occurs once for a container, using the rowID of the first column partition value as the rowID of the entire container, instead of storing a row header for each column partition value. Vantage can determine the rowid of a column partition value by its position within the container.

If many column partition values can be packed into a container, row header compression can greatly reduce the space needed for a column-partitioned object compared to the storage required for the same object without column partitioning.

If only a few column partition values can be packed in a container because of their width, it is possible for there to be a small increase in the space needed for a column-partitioned table or NoPI join index compared to the space required by the object without column partitioning. In this case, ROW format may be more appropriate than COLUMN format for storing the object.

If only a few column partition values can be stored using ROW format because the row partitioning is such that only a few column partition values occur for each combined partition, it is possible for there to be a very large increase in the space required to store a column-partitioned object compared to the space required for the same object without column partitioning. In the worst case, the space required to store a column-partitioned object can increase by up to nearly 24 times.

If this occurs, consider one of the following alternatives.

- Alter the row partitioning to produce more column partition values per combined partition.
- Remove column partitioning from the object definition.

Autocompression

When you create a column-partitioned table, Vantage compresses the data that you insert into the physical table rows unless you specify the NO AUTO COMPRESS option for the table or for specific column partitions.

Unless you specify NO AUTO COMPRESS, Vantage does the following for column partitions with the COLUMN format:

- Considers the cost of decompressing the data and determines whether autocompression is worthwhile
- Selects a compression method
- Automatically compresses values and decompresses them when you need to retrieve them

Autocompression is most effective for a column partition with a single column and COLUMN format.

Vantage determines whether to apply both autocompression and user-specified multivalue and algorithmic compression to a column, and if the user-specified methods do not help to compress a container, the system only applies autocompression.

Using Hash Index and Join Index Row Compression

Row compression automatically divides rows into repeating and nonrepeating portions. The system appends multiple sets of nonrepeating column values to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical extensions of the base repeating set. The nonrepeating column set has a pointer to each of the repeating column sets so it can reconstruct the rows when required.

Row compression significantly reduces index storage space for index columns that have a large number of repeating values.

For hash indexes, the system compresses rows by default.

For join indexes you can:

1. Specify the columns to be compressed using the *(column_1_name)* syntax elements in the SELECT clause of a CREATE JOIN INDEX statement.
2. Specify the columns to remain uncompressed using the *(column_2_name)* syntax elements in the SELECT clause for the same CREATE JOIN INDEX statement.

Example 1: Join Index Row Compression

Assume that a join index specifies 5 columns:

- Two columns with repeating values:
 - Column A indicates whether the customer owns a truck or car
 - Column B indicates whether the vehicle is domestic or foreign made
- Three columns of unique, or mostly unique, information:
 - Column C is the customer name
 - Column D is the customer address
 - Column E is the customer phone number

The uncompressed join index row data is as follows:

```
(car,foreign,joe,addr1,ph1)
(car,foreign,jane,addr2,ph2)
(car,foreign,max,addr3,ph3)
(car,domestic,bill,addr6,ph6)
(car,domestic,linda,addr7,ph7)
(truck,domestic,brad,addr4,ph4)
(truck,domestic,sam,addr5,ph5)
(truck,foreign,dave,addr8,ph8)
(truck,foreign,rick,addr9,ph9)
```

You can specify row compression in a CREATE JOIN INDEX statement, for example:

```
CREATE JOIN INDEX  ji_name  AS
    SELECT (col_a,col_b),(col_c,col_d,col_e)
    FROM  table_name
    . . . ;
```

where the first set of parentheses defines the columns (*col_a* and *col_b*) for which duplicate row values are compressed, and the second set of parentheses defines the remaining columns (*col_c*, *col_d*, and *col_e*), for which row values are not compressed.

As a result, the system stores each possible combination of the repeating values for *col_a* and *col_b* once, grouped with the related row values for the uncompressed columns, for example:

```
((car,foreign),(joe,addr1,ph1),(jane,addr2,ph2),(max,addr3,ph3))
((car,domestic),(bill,addr6,ph6),(linda,addr7,ph7))
((truck,domestic),(brad,addr4,ph4),(sam,addr5,ph5))
((truck,foreign),(dave,addr8,ph8),(rick,addr9,ph9))
```

Benefits

Row compression significantly reduces index storage space for index columns that have a large number of repeating values.

Usage Notes

Row compression is valuable where column values repeat from row to row. Columns in a join index that have a high percentage of distinct (non-repeating) values are not a good candidates for row compression. Row compression cannot be used with data using row-level security constraints. A join index with row compression cannot be column-partitioned.

In addition to any row compression you define for a join index, the index also inherits MVC compression defined for its base tables, with some exceptions.

Related Information

For information on...	See...
strategies and restrictions when specifying row compression in a join index, including how join indexes inherit MVC from base tables	CREATE JOIN INDEX information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184
syntax and examples for using the CREATE JOIN INDEX statement	CREATE JOIN INDEX information <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
join index row compression as it relates to database design	Join Index Storage

Block-Level Compression

A data block is a physical unit of I/O that defines how Vantage handles data. When you specify block-level compression (BLC), Vantage stores data blocks in compressed format to save storage space. BLC can be enabled or disabled system-wide or for specific types of tables only. BLC can be applied to these objects:

- Journal tables
- Join indexes
- Hash indexes
- Spool tables
- Global temporary tables
- Primary subtables
- Fallback subtables
- LOB subtables (JSON, XML, and character)

Block-level compression is independent of any other type of compression applied to the same data.

The goals of block-level compression are to save storage space and to reduce disk I/O bandwidth. Block-level compression can use significantly more CPU to compress and decompress data dynamically, so whether query performance is enhanced with block-level compression depends on whether performance is more limited by disk I/O bandwidth or CPU usage.

Teradata offers software-based and hardware-based BLC. You select the BLC method when you specify the compression algorithm in the CompressionAlgorithm field in DBS Control. See [Controlling BLC at the System Level Using the DBS Control Utility](#).

The following site provides guidelines for evaluating the impact of block-level compression on performance and on disk space: <https://downloads.teradata.com/extensibility/articles/block-level-compression-evaluation-with-the-blc-utility>. The site also provides a link to download the Block Level Compression Evaluation utility.

Hardware-Based Block-Level Compression

Hardware-based BLC is available only on Teradata systems that include compression hardware. Hardware-based BLC creates little or no CPU resource contention with other database operations. Hardware-based block-level compression provides the same functions as software-based block-level compression.

Note:

Systems configured with BLC hardware also contain backup software that uses the same compression algorithm as the hardware. If the system is using hardware BLC and Vantage detects that the compression hardware is not functioning, the backup software automatically performs the compression.

To specify use of compression hardware, set the CompressionAlgorithm field in the DBS Control utility to ELZS-H.

Coexistence of Software BLC and Hardware BLC

Vantage lets data compressed with hardware compression and data compressed with software compression coexist simultaneously in the same table. This allows you to change easily from one method to the other.

Software-Based Block-Level Compression

Software-based BLC is designed for use on large volumes of seldom-accessed data. This is because access requires CPU-intensive decompression. BLC can also achieve relatively good compression on unknown/unclassified data across different column types, including the internal data structures and the row header. Teradata recommends that you do not load or access tables that use software-based BLC during peak hours (except on CPU-rich platforms).

Block-level compression can be enabled at the system level using the DBS Control utility and at the table level using SQL DDL.

Enabling BLC and Temperature-Based BLC at the System Level

To enable BLC for the system, use the DBS Control setting BlockLevelCompression. To enable temperature-based BLC for the system, enable both the BlockLevelCompression and EnableTempBLC settings.

About Temperature-Based Block-Level Compression

You can specify that data is automatically compressed if it is COLD (infrequently accessed) and automatically decompressed when it becomes WARM, HOT, or VERYHOT (more frequently accessed). By default, COLD data is the 20% of the data that is the least often accessed, and HOT and VERYHOT data is the 20% of the data that is the most often accessed. WARM data is anything in between.

Enabling Compression for Individual Tables Using DDL

Enabling block-level compression at the system level allows you to enable block-level compression for individual tables. To accomplish this, use the optional `BLOCKCOMPRESSION = block_compression_option` clause for the statements `CREATE TABLE`, `ALTER TABLE`, `CREATE HASH INDEX`, and `CREATE JOIN INDEX`. If you do not use a `BLOCKCOMPRESSION` clause or choose the `DEFAULT` keyword, the DBS Control DefaultTableMode is used. The valid specifications for *block_compression_option* in these statements are:

DDL Statement BLOCKCOMPRESSION Keywords	Result
MANUAL	Table data is not compressed except when: <ul style="list-style-type: none"> You specify a query band option to compress the table. You use the Ferret COMPRESS command. The DBS Control settings enable BLC for specific tables or table types. In this case, the DBS Control setting cannot be NEVER.
AUTOTEMP	Table data is compressed or decompressed automatically based on the data temperature.
ALWAYS	Data for the table is compressed, even if query band options or DBS Control utility fields indicate otherwise.
NEVER	The table is never compressed, even if DBS Control settings, query band options, or Ferret commands indicate otherwise.
DEFAULT	The DBS Control tunable DefaultTableMode determines whether the table is manually, automatically, or never compressed. A table set to DEFAULT can be affected by any future change to the system default DBS Control setting.

Setting the Compression Algorithm for Individual Tables Using DDL

To specify the compression algorithm when you create a table, join index, or hash index or alter a table, use the optional `BLOCKCOMPRESSIONALGORITHM` option. Possible `BLOCKCOMPRESSIONALGORITHM` values include the following:

DDL Statement BLOCKCOMPRESSIONALGORITHM Keywords	Result
ZLIB	The ZLIB software algorithm is used to compress data blocks.
ELZS_H	The ELZS_H hardware algorithm is used to compress data blocks on systems configured with a hardware board.
DEFAULT	The compression algorithm is controlled by the setting of the DBS Control utility field CompressionAlgorithm.

Setting the Software Compression Level for Individual Tables Using DDL

To specify the software compression level when you create a table, join index, or hash index or alter a table, use the optional `BLOCKCOMPRESSIONLEVEL` option. Possible `BLOCKCOMPRESSIONLEVEL` values include the following:

DDL Statement <code>BLOCKCOMPRESSIONLEVEL</code> Keywords	Result
1-9	The compression level, where 1 indicates greatest compression speed and 9 indicates greatest compression possible.
DEFAULT	The compression level is controlled by the setting of the DBS Control utility field <code>CompressionLevel</code> .

Restrictions and Usage Notes

The `BLOCKCOMPRESSION` option applies only to permanent tables (except permanent journal tables).

To apply block-level compression to permanent journals and tables that do not survive restarts, set the DBS Control utility fields that apply to them. See [Controlling BLC at the System Level Using the DBS Control Utility](#).

When you change the value of `BLOCKCOMPRESSION`, Vantage does not change the compression status of data blocks that existed before the change. This means that some data blocks in the table may be uncompressed and some compressed. Changing the value of `BLOCKCOMPRESSION` affects only new data blocks. To make the table consistent, use the `FERRET [UN]COMPRESS` commands.

The `BLOCKCOMPRESSION` option cannot be used with the `IMMEDIATE` clause.

Compressing Tables, Databases, and Data Types Manually Using Ferret

Use the Ferret utility command `COMPRESS` to compress:

- Data blocks of an existing table.
- All tables in a database or all data of a particular type in a database: Primary, fallback, primary compressible LOB (JSON, XML, and character LOB), and fallback compressible LOB (JSON, XML, and character LOB).

After a table is compressed, rows added to the table are compressed automatically. To uncompress the data blocks manually, use the Ferret command `UNCOMPRESS`.

To estimate the effects of the `COMPRESS` or `UNCOMPRESS` commands before you run them, use the `ESTIMATE` option. The output, which is based on sampling your data, shows the estimated block size and CPU usage that will result if you run the command.

Usage Notes

- Teradata does not recommend using the Ferret COMPRESS/UNCOMPRESS commands on a table using temperature-based BLC because if the compressed state of the data does not match its temperature, the system may undo the commands.
- If you use a COMPRESS command on a table specified as BLOCKCOMPRESSION=NEVER, the command is rejected.
- If you use an UNCOMPRESS command on a table specified as BLOCKCOMPRESSION=ALWAYS, the command is rejected.
- Secondary index primary and fallback subtables are not compressed, regardless of DBSControl Compression settings, table-level attributes, and the BlockCompression query band.

Compressing Data Loaded into Empty Tables Set to MANUAL

To compress data you are loading into an empty permanent or global temporary table that has the MANUAL compression setting, specify BlockCompression in a SET QUERY_BAND statement:

```
SET QUERY_BAND = 'BlockCompression=YES;' FOR SESSION;
```

Note:

If you set BlockCompression to a value other than the keywords in the following table, the BlockCompression value is not valid and the session still has the default DBS Control settings.

Possible values for BlockCompression, which are applicable to session and transaction query bands, include:

BlockCompression SET QUERY_BAND Keywords	Result
YES	Compress all primary table data, primary compressible LOBs (JSON, XML, and character LOBs), fallback table data, and fallback compressible LOBs (JSON, XML, and character LOBs).
NO	Do not compress any new data.
ALL	Compress all primary table data, primary compressible LOBs (JSON, XML, and character LOBs), fallback table data, and fallback compressible LOBs (JSON, XML, and character LOBs). Same effect as YES.
NONE	Do not compress any data. Same effect as NO.
FALLBACK	Compress fallback table data and fallback compressible LOBs (JSON, XML, and character LOBs). Note: You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

BlockCompression SET QUERY_BAND Keywords	Result
ONLYCLOBS	Compress primary and fallback compressible LOBs (JSON, XML, and character LOBs).
WITHOUTCLOBS	Compress all data except primary and fallback compressible LOBs (JSON, XML, and character LOBs).
FALLBACKANDCLOBS	Compress fallback table data and primary and fallback compressible LOBs (JSON, XML, and character LOBs).

Note:

Secondary index primary and fallback subtables are not compressed, regardless of DBSControl Compression settings, table-level attributes, and the BlockCompression query band.

The BlockCompression query band setting can override these DBS Control settings:

- The default system-level compression (block-level) settings.
- CompressPermPrimaryDBs and CompressPermFallbackDBs.
- CompressGlobalTempPrimaryDBs and CompressGlobalTempFallbackDBs.
- CompressMLoadWorkDBs.

The BlockCompression query band setting cannot override a table-level setting of ALWAYS or a system-level setting of NEVER.

Compressing Data Loaded into Empty Subtables Set to AUTOTEMP

To assign different temperatures to data loaded in empty primary, fallback, and compressible LOB subtables (XML, JSON, and character LOB) that have the AUTOMATIC setting, use the SET QUERY_BAND statement choosing the temperature values VERYHOT, HOT, WARM, or COLD for the following:

- TVSTEMPERATURE_PRIMARY: The temperature to assign to the primary row subtable, the primary row secondary index subtables, and the primary LOB subtables (excluding compressible LOBs).
- TVSTEMPERATURE_PRIMARYCLOBS: The temperature to assign to the primary row compressible LOB subtables.
- TVSTEMPERATURE_FALLBACK: The temperature to assign to the fallback row subtable, the fallback row secondary index subtables, and the fallback LOB subtables (excluding compressible LOBs).
- TVSTEMPERATURE_FALLBACKCLOBS: The temperature to assign to the fallback row compressible LOB subtable.
- TVSTEMPERATURE: The temperature to assign to all the subtables in the preceding bullets.

For example:

```
SET QUERY_BAND = 'TVSTEMPERATURE_FALLBACK = COLD;' FOR SESSION;
```

This indicates that the fallback data should be cold and the other subtables should use the system default temperature for the table type (as defined in DBS Control).

An example of using multiple specifications is:

```
SET QUERY_BAND = 'TVSTEMPERATURE_PRIMARY=HOT; TVSTEMPERATURE = COLD; '
FOR SESSION;
```

This indicates that the primary row subtable should be hot and all other subtables should be cold. In this example, the TVSTEMPERATURE name takes precedence over any system default temperature settings.

You can specify multiple TVSTEMPERATURE-style names if the exact name is used only once.

There are a number of advantages to managing compression based on the data temperature.

- Large sets of historical data that are not frequently accessed are often kept uncompressed.
In some cases, you might not even realize that data is no longer being used. Temperature-based compression automatically locates such data and compresses it.
- Data in partitions of partitioned tables are often not frequently used.
Temperature-based compression enables you to compress such partition data automatically.
- By using temperature-based compression with date-partitioned tables where new partitions are constantly being added, but old partitions are retained, you are relieved of determining which partitions are old enough to have become infrequently used.

For more information, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Note:

In most cases, the data will be assigned the temperature you specify and will be moved more quickly to appropriate storage media. However, in cases where newly loaded table data is stored on existing cylinders that also store data from other tables, the temperature and storage location of those cylinders will not be changed.

Interaction between Block-Level Compression and TVS Temperature Query Band Values

If you set both TVSTEMPERATURE and BLOCKCOMPRESSION values for a query band, unless the table setting for BLOCKCOMPRESSION is NEVER, the result is:

- BLOCKCOMPRESSION determines the compression if the table setting is MANUAL.
- TVSTEMPERATURE determines the compression if the table setting is AUTOTEMP.

If the setting for BLOCKCOMPRESSION is ALWAYS, the table is always compressed.

Controlling BLC at the System Level Using the DBS Control Utility

The DBS Control utility applies BLC at the system level, enabling or disabling compression globally, or for specific types of tables (for example, permanent tables or global temporary tables). Changes to DBS Control compression (block-level) settings have no effect on existing data.

The following tables briefly describe the DBS Control BLC system-level fields. For further information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

System-Level Fields

The following DBS Control fields control the compression of data blocks at the system level.

DBS Control BLC System-Level Fields	Description
BlockLevelCompression	<p>Indicates whether the block-level compression feature is enabled. Other compression settings may separately enable or disable BLC for specific tables or table types.</p> <p>Note: BLC is disabled by default. When BlockLevelCompression is off, that includes temperature-based block-level compression. Disabling BLC also disables the Ferret COMPRESS command.</p>
EnableTempBLC	<p>Enables temperature-based block-level compression. If this value is enabled, Teradata recommends that the DBS Control field DisableAutoCylPack be set to FALSE and AutoCylPackColdData be set to TRUE.</p> <p>If you disable this value while BlockLevelCompression is enabled, tables using automatic temperature-based compression use MANUAL compression.</p> <p>At any time, different portions of a table, such as different partitions, may exist in different states of compression depending on their temperatures.</p>
DefaultTableMode	<p>Determines how BLC is managed for permanent tables that have the BLOCKCOMPRESSION option set to default or that do not have the BLOCKCOMPRESSION option specified when the table is created. The valid settings follow.</p> <p>MANUAL: BLC is applied based on the settings of these DBS Control fields, which determine the default BLC for these table types at the time the table is created:</p> <ul style="list-style-type: none"> • CompressPermPrimaryDBs • CompressPermFallbackDBs • CompressPermPrimaryCLOBDBs • CompressPermFallbackCLOBDBs <p>Tables can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands.</p> <p>Data inserted into existing tables inherits the current compression status of the table at the time the data is inserted.</p>

DBS Control BLC System-Level Fields	Description
	<p>Vantage manages spool, redrive, MultiLoad work, and permanent journal tables manually only, even if DefaultTableMode is set to NEVER or AUTOTEMP. See the descriptions of CompressSpoolDBs, CompressMloadWorkDBs, and CompressPJDBs in this table.</p> <p>AUTOTEMP: BLC is automatically performed by Vantage based on the frequency of access to the table data (the data temperature). Temperature-based BLC (TBBLC) compresses infrequently accessed data, and uncompresses data that it is accessed frequently.</p> <p>Note:</p> <p>BLC is automatically performed by Vantage based on the frequency of access to the table data (the data temperature). Temperature-based BLC (TBBLC) compresses infrequently accessed data, and uncompresses data that it is accessed frequently.</p> <p>ALWAYS: Table data is always compressed at the data block level.</p> <p>NEVER: Table data is never compressed at the data block level.</p>
MinDBSectsToCompress	Specifies the minimum size data block to compress. Data blocks that do not meet the minimum size requirement are not compressed. The valid range is 2 through 255 sectors. The default is 16.
MinPercentCompReduction	Specifies the minimum percentage by which the size of a data block must be reduced by compression. If compression cannot reduce the data block by at least this amount, the data block is not compressed. The valid range is 0 through 99 percent. The default is 20 percent.
CompressionAlgorithm	<p>Specifies the algorithm used to compress and uncompress data blocks and the compression method used (software or hardware-based):</p> <ul style="list-style-type: none"> • ZLIB (Lempel-Ziv algorithm) Standard, software-based compression, the default • ELZS_H (Exar Lempel-Ziv_Stac algorithm) Hardware-based compression available only on systems with compression engine hardware installed on every node. • ELZS_S (Exar Lempel-Ziv_Stac algorithm) Software equivalent of hardware compression. <p>Note:</p> <p>If the setting is ELZS_H and the compression hardware malfunctions, the system automatically defaults to using ELZS_S. Use of ELZS_S is not recommended unless the hardware compression board is malfunctioning.</p>
CompressionZLIBMethod	<p>Specifies the implementation of the ZLIB algorithm to use for compressing and uncompressing data blocks.</p> <ul style="list-style-type: none"> • ZLIB (Lempel-Ziv algorithm). Standard ZLIB library. • IPPZLIB (Lempel-Ziv algorithm using optimization from the Intel Integrated Performance Primitives (IPP) library). This is the default. This implementation provides better compression performance on Intel platforms.

DBS Control BLC System-Level Fields	Description
CompressionLevel	<p>Determines whether compression operations favor processing speed or degree of data compression.</p> <p>The valid range is 1 through 9. A setting of 1 favors compression speed over degree of data compression. A setting of 9 favors degree of data compression over compression speed. The default is 6, which provides a balance.</p> <p>Note:</p> <p>CompressionLevel affects only software-based block-level compression. It does not apply to hardware-based compression or its equivalent software.</p>
UncompressReservedSpace	<p>Specifies the minimum percentage of storage space that must remain available while DBs are uncompressed using the Ferret UNCOMPRESS command.</p> <p>The UNCOMPRESS operation terminates if the threshold is exceeded. Set this field to represent the expected peak amount of required spool space. The valid range is 1 through 90%. The default is 20%.</p>
OverrideARCBLC	<p>Determines whether database tables restored from archives have their data blocks compressed according to the effective BLOCKCOMPRESSION setting of each table when it is archived or according to the system-level BLC defaults when the tables are restored.</p> <p>Note:</p> <p>This field does not affect tables with effective BLOCKCOMPRESSION values of ALWAYS or NEVER or DEFAULT if the DefaultTableMode is set to ALWAYS or NEVER.</p> <p>Block compression that is controlled by the BLOCKCOMPRESSION query band takes precedence over the OverrideARCBLC field setting.</p> <p>Valid settings include:</p> <ul style="list-style-type: none"> • TRUE: The system default settings for block-level compression are used to determine whether restored tables are block-level compressed. • FALSE: The effective table BLOCKCOMPRESSION value at the time of archiving determine whether restored tables are block-level compressed. <p>This field does not affect subtables for which the effective BLOCKCOMPRESSION is set to ALWAYS or NEVER on the target system to which the data is restored.</p> <p>For permanent tables and global temporary tables, different subtables hold primary data, fallback data, and LOB data. BLC for these categories of data can be independently controlled using DBS Control Compression fields 14 - 21 (Compress...DBs fields). These subtable settings are effective only when the BLOCKCOMPRESSION setting for the corresponding base table is set to MANUAL, either explicitly, or as a result of being set to DEFAULT while the DefaultTableMode DBS Control field is set to MANUAL.</p>

Fields Specific to Certain Types of Subtables

The following DBS Control fields control the compression of data blocks for specific types of subtables.

BLC Fields Specific To Certain Types of Tables	Description
CompressPermPrimaryDBs	Specifies if and when primary subtable data in permanent storage is compressed. This field applies only to tables using MANUAL compression.
CompressPermFallbackDBs	Specifies if and when fallback subtables in permanent storage are compressed. This field applies only to tables using MANUAL compression.
CompressPermPrimaryCLOBDBs	Specifies if and when primary compressible LOB (JSON, XML, character LOB) subtable data in permanent storage is compressed. This field applies only to tables using MANUAL compression.
CompressPermFallbackCLOBDBs	Specifies if and when fallback compressible LOB (JSON, XML, character LOB) subtable data in permanent storage is compressed. This field applies only to tables using MANUAL compression.
CompressSpoolDBs	Specifies the conditions under which spool, volatile, and redrive data blocks are compressed. Valid values include: <ul style="list-style-type: none"> • ALWAYS: The data blocks of a new spool table will be compressed. • NEVER: The data blocks of a new spool table will not be compressed. This is the default. The Ferret utility COMPRESS command will have no effect. • IFNOTCACHED: The DBs of a new table will be compressed if they are not being cached.
CompressMloadWorkDBs	Specifies conditions under which data blocks from MultiLoad sort worktables and index maintenance worktables are compressed. MultiLoad worktables are temporary tables built during the acquisition phase of loading.
CompressPJDBs	Specifies whether all permanent journal DBs are compressed. Valid values are: <ul style="list-style-type: none"> • ALWAYS: The DBs of a new PJ table will be compressed. • NEVER: The DBs of a new PJ table will not be compressed. The Ferret utility COMPRESS command will have no effect.
CompressGlobalTempPrimaryDBs	Specifies if and when primary data subtables in global temporary storage are compressed.
CompressGlobalTempFallbackDBs	Specifies if and when fallback data subtables in global temporary storage are compressed.
CompressGlobalTempPrimaryCLOBDBs	Specifies if and when primary compressible LOB (JSON, XML, character LOB) subtables in global temporary storage are compressed.
CompressGlobalTempFallbackCLOBDBs	Specifies if and when fallback compressible LOB (JSON, XML, character LOB) subtables in global temporary storage are compressed.

Note:

Secondary index primary and fallback subtables are not compressed, regardless of DBSControl Compression settings, table-level attributes, and the BlockCompression query band.

Valid Settings

Unless otherwise specified above, the valid settings for the preceding DBS Control fields are:

Valid Setting	Description
ALWAYS	The data blocks of a new table will be compressed regardless of query band options specified at load time.
ONLYIFQBYES	The data blocks of a new table will not be compressed unless a query band used at load time specified to compress them.
ONLYIFQBNO	The data blocks of a new table will be compressed unless a query band used at load time specified not to compress them.
NEVER	The data blocks of a new table will not be compressed regardless of any query band used at load time. The Ferret utility COMPRESS command will have no effect.

Note:

Compression is subject to the BlockLevelCompression field setting, and to the criteria specified by the MinPercentCompReduction and MinDBSectsToCompress DBS Control fields. Data blocks not meeting these minimum criteria will not be compressed.

Temperature-Based BLC Fields

The following DBS Control fields control temperature-based block-level compression.

Temperature-based Compression Fields	Description
TempBLCThresh	Indicates the temperature threshold at which data is compressed in tables using temperature-based compression. Data at or colder than the specified temperature is compressed. Temperatures can be specified by keywords COLD, WARM, HOT, VERYHOT, or a percentage of the coldest allocated user data that should be compressed.
TempBLCSpread	Exempts data within a user-specified percentage of the threshold TempBLCThresh from triggering automatic compression or decompression. Use this value to prevent data whose temperature is close to the threshold from being repeatedly compressed and uncompressed. For example, if the TempBLCThresh is defined as COLD and TempBLCSpread is 5%, then the data must be 5% colder than COLD to be compressed and 5% warmer than COLD to be uncompressed.

Temperature-based Compression Fields	Description
TempBLCInterval	Specifies the time to wait after temperature-based compression/decompression before checking if other data needs to be compressed or decompressed.
TempBLCIOThresh	Specifies the maximum number of I/Os on a node before the autocompression background task AutoTempComp sleeps for a short interval. This setting reduces the performance impact of temperature-based compression on foreground workloads. If the system remains busy, the task will sleep according to TempBLCInterval.
TempBLCPriority	The priority level of AutoTempComp, the background task that automatically compresses and decompresses data. Valid values are: LOW (or BOTTOM), MEDIUM (or DEFAULT), HIGH, or RUSH (or TOP).
TempBLCRescanPeriod	The number of days before the compression of cylinders using temperature-based compression is revalidated. This rescan is necessary because the temperature of a cylinder can change. At least one scan is performed after each startup.

Block-Level Compression Usage Notes

Teradata recommends that when you are using BLC that you set the CREATE TABLE or ALTER TABLE DATABLOCKSIZE option for each affected table to the maximum setting for your system. Specifying the maximum DATABLOCKSIZE for a table results in effective compression with the fewest compressed data blocks and the fewest required compression/decompression operations. For more information on large data blocks, see *1 MB Data Blocks in Teradata® Orange Book*, 541-0010379.

Block-level compression can result in some operations using considerably more CPU while operating on compressed tables (for example, queries, insert/updates, archive and restores, and the Reconfiguration and CheckTable utilities). Unless the system is very CPU rich, these operations will impact other workloads and could lengthen elapsed response times.

Use BLC only for large tables, for example, those tables which, in uncompressed form, are more than 5 times as large as system memory. Although you can use BLC on smaller tables, the CPU cost may outweigh the space benefits, depending on your system load and capability.

BLC can reduce the I/O demand for I/O-intensive DSS queries on compressed tables. This may be useful in situations where CPU is available for the decompression and workload management can keep the I/O intensive DSS queries to an appropriate level of consumption.

Note:

To restore a Data Stream Architecture archive made on a source system with hardware-based block-level compression, install the driver package for the hardware compression cards, `teradata-expressdx`, on the target system. Do this even if the target is not set up for hardware compression, so the target can read the compressed archive.

Determining How Often Data Is Used

To understand how often data is used in tables and other database objects on systems with TVS or Temperature-Based Block-Level Compression, use the Heatmap table function, `tdheatmap`. You can use this function for the following purposes:

- Compare the relative temperatures of tables or cylinders over time
- Determine which tables or cylinders are targeted to the Teradata Intelligent Memory VERYHOT cache
- Gain information about where data is stored, including media type, storage class, and storage grade (SLOW, MEDIUM, or FAST)

For more information about the Heatmap table function, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Interaction between Block-Level Compression Settings

The order of precedence between coexisting block-level compression settings is the following:

- The DBS Control utility setting applies when there are no other settings.
- The query band setting overrides the DBS Control utility setting.
- The table-level setting overrides everything else.

CPU Considerations on Different Teradata Platforms

Because software-based BLC is CPU-intensive, consider the CPU capabilities of the particular Teradata platform when using it. When using BLC on Teradata appliance systems with a very high CPU capacity, you can:

- Apply BLC to all allowable types of data.
- Load and access data at any time, because there is enough available CPU to frequently decompress data.

When using software-based BLC on other platforms with less CPU capacity, you may need to:

- Load tables during off-peak hours.
- Limit access to compressed tables during critical throughput periods.

Some operations (including queries, insert/updates, dump/restores, reconfig, and CheckTable) can use considerably more CPU while operating on compressed tables. Unless the system is very CPU rich, these operations will impact other workloads and could lengthen elapsed response times.

Finding Tables and Databases with Compressed Data Blocks

To find tables that use BLC, issue the Ferret command `SHOWCOMPRESS`. The command lists the names of tables with compressed BLC data blocks, whether they were compressed manually or automatically with temperature-based compression.

Obtaining Information about Tables with Compressed Data Blocks

Use different methods to obtain different types of information about block-level compressed tables.

Estimated Space Savings Percentage for Manually Compressed Tables

To see the estimated space savings percentage for manually compressed BLC tables, issue a SELECT request on the BLCCompRatio column of the DBC.StatsV, DBC.TableStatsV, or DBC.TempTableStatsV views. For example:

```
SELECT blccompratio from dbc.statsv WHERE tablename = 'cvis001';
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
  BLCCompRatio
  -----
             86
```

File System Information in an SQL Table

To see Ferret SHOWBLOCKS-like file system information in an SQL table, use the CreateFsysInfoTable and PopulateFsysInfoTable macros.

The following example creates a volatile table to hold a small display of disk block information:

```
exec
dbc.createfsysinfotable('targetdatabasename','targettablename','volatile','showblo
cks','s');
```

The next example populates that volatile table with data block histogram and block-level compression details for the specified input table:

```
exec
dbc.populatefsysinfotable('inputdatabasename','inputtablename','showblocks','s','
targetdatabasename','targettablename');
```

For more information, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Detailed Compression Statistics for Tables

To see detailed compression statistics, including the percentage of a table that is compressed, use a SHOW STATISTICS VALUES request. An XML version is also available. For example:

```
show summary statistics values on cvis300;
*** Text of DDL statement returned.
```

```

*** Total elapsed time was 1 second.
-----
COLLECT SUMMARY STATISTICS
      ON Albee.cvis300
      VALUES
(
  /** TableLevelSummary **/
  /* Version                */ 6,
  /* NumOfRecords           */ 1,
  /* Reserved               */ 0.000000,
  /* Reserved               */ 0.000000,
  /* CurrSysInsertCnt       */ -1,
  /* CurrSysDeleteCnt      */ -1,
  /* CurrSysInsDelResetTS   */ TIMESTAMP '9999-12-31 23:59:59-00:00',
  /* NumOfCurrSysUpdateCols*/ 0,
  /* SummaryRecord[1]      */
  /* Temperature           */ 0,
  /* TimeStamp             */ TIMESTAMP '2014-11-05 23:00:38-00:00',
  /* NumOfAMPs             */ 4,
  /* OneAMPsSampleEst      */ 2092,
  /* AllAMPsSampleEst      */ 2048,
  /* RowCount              */ 2048,
  /* DelRowCount           */ 0,
  /* PhyRowCount           */ 2048,
  /* AvgRowsPerBlock       */ 0.992188,
  /* AvgBlockSize (bytes)  */ 9216.000000,
  /* BLCpctCompressed      */ 100.00,
  /* BLCBlkUcpuCost (ms)   */ 0.400000,
  /* BLCCompRatio          */ 86.000000,
  /* AvgRowSize            */ 120.000000,
  /* Reserved              */ 0.000000,
  /* Reserved              */ 0.000000,
  /* Reserved              */ 0.000000,
  /* StatsSkipCount        */ 0,
  /* SysInsertCnt          */ 0,
  /* SysDeleteCnt          */ 0,
  /* SysUpdateCnt          */ 0,
  /* SysInsDelLastResetTS  */ TIMESTAMP '9999-12-31 23:59:59-00:00'
);

```

Estimated Compression Ratio

To see how much of a table is compressed (the estimated compression ratio), use the Ferret SHOWBLOCKS command. For example, the following command shows BLC information for primary data subtables:

```
SHOWBLOCKS /S
```

For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Statistics, Including Estimated Compression Ratio

To calculate statistics including the average block size and estimated compression ratio for tables compressed manually with BLC, issue a COLLECT STATISTICS or COLLECT STATISTICS SUMMARY request. For example, this request collects summary statistics on the employee table:

```
COLLECT SUMMARY STATISTICS ON employee;
```

For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Considerations for Choosing a Method To Obtain BLC Information

For smaller tables that may not have data on all AMPs, Teradata recommends using the Ferret SHOWBLOCKS command instead of a COLLECT STATISTICS or COLLECT SUMMARY STATISTICS request. The reason is that the COLLECT STATISTICS estimate is based on sample data from a single AMP, while the SHOWBLOCKS estimate is based on a larger sample from all AMPs. The output of SHOWBLOCKS is more reliably accurate for smaller tables.

Note:

The SHOW STATISTICS VALUES and COLLECT STATISTICS SUMMARY requests provide BLC data about manually compressed tables only. They do not provide BLC data on tables automatically compressed with temperature-based compression.

About System and AMP Outages During Compression

Block-level compression is fault-tolerant:

- If an AMP is down, the Ferret COMPRESS and UNCOMPRESS commands can still operate.
- If a database restart occurs when a table is being compressed or uncompressed, the interrupted operation completes during system recovery.

To see the compression or uncompression operations in progress during a system recovery after a database restart, use the Ferret command SHOWAMPRECOVERYBLC. You can also use that command to cancel the operations you just viewed if you want to minimize transaction recovery time after a restart.

Note:

- Compress and uncompress operations that are canceled during recovery via the SHOWAMPRECOVERYBLC command will remain in a mixed compressed and uncompressed state. Run the Ferret COMPRESS or UNCOMPRESS command again to make the table state consistent.
- If a database restart occurs when all database tables are being compressed or uncompressed, the interrupted operation completes only on the table in-progress at the time of the restart.

CPU Costing for Software-Based BLC

Software-based BLC is a space-saving feature and rarely helps performance. The following table shows examples of relative CPU cost per MB to compress or decompress data, assuming a large table and full-table scans. Increasing the amount of joins and indexes, or the retrieval spool used by executing queries, reduces the amount of compression I/O.

Platform	CPU Cost (CPU msec/MB)		Uncompressed I/O (MB/sec/node)		Compressed I/O (MB/sec/node)	
	Compress	Uncompress	Compress	Uncompress	Compress	Uncompress
2500/5500	50.0 - 100.0	7.0 - 14.0	40 - 80	364 - 571	13 - 27	121 - 190
1550/2550/ 5550	63.9 - 127.8	8.9 - 14.1	63 - 125	569 - 894	21 - 42	190 - 298
1600/2580/ 5600	73.8 - 147.5	10.3 - 16.2	108 - 217	986 - 1550	36 - 72	329 - 517
2650/5650	73.8 - 147.5	10.3 - 16.2	163 - 325	1479 - 2324	54 - 108	493 - 775

Estimating BLC Space Savings and CPU Usage

Use the ESTIMATE option for the Ferret COMPRESS or UNCOMPRESS commands to see the estimated block size and CPU usage that will result if you run the command. The results are based on sampling your data.

Choosing a Software-based Compression Scheme

In some cases, such as when column values are mostly unique, algorithmic compression can provide better compression results than multivalue compression. Because multivalue compression can coexist with algorithmic compression, users have the flexibility of an IF-ELSE option: either compressing values from a dictionary or using a custom compression algorithm.

Algorithmic compression has a fairly high overhead each time the compressed column is accessed during the execution of a query. It is best used for columns that are used only for the final projected answer set.

Combining Compression Methods

You can use more than one compression method on the same data set. The effects are generally additive, but overall compression will be somewhat less than the total of individual results.

Block-level compression combines well with multivalue compression. Teradata does not recommend using algorithmic compression at the same time as block-level compression due to performance concerns.

Related Information

Information on...	Is available in...
specifying MVC, ALC, or BLC compression in a CREATE TABLE or ALTER TABLE statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
specifying join index row compression in a CREATE JOIN INDEX statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
using the Ferret utility COMPRESS and UNCOMPRESS commands to apply BLC to populated tables	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102
using the DBS Control utility to set or modify BLC parameters	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102
using the BlockCompression query band in a SET QUERY BAND statement to override BLC settings	<i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184
applying compression to various data types using the COMPRESS and DECOMPRESS phrases	<i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143
block-level compression use cases and examples	<i>Block-Level Compression Orange Book</i> , TDN0001167

Database-Level Capacity Planning Considerations

This section describes some of the database-level considerations for capacity planning.

[System-Level Capacity Planning Considerations](#) describes file system- and hardware-oriented considerations for capacity planning.

Capacity Planning

When there is no legacy database to build on, capacity planning can be a difficult enterprise to undertake. Fortunately, this is rarely an issue for contemporary users because at least part of their corporate databases are almost always maintained in electronic form. Keep in mind that much of the information presented in this section assumes you have a legacy system to draw upon for making your sizing estimations.

Capacity planning should begin with the idea of making the most frequently accessed data available at all times. With the relatively low priced, large capacity disk storage units commonly used for data warehousing applications, the nature of the emphasis on this factor has changed from offloading as much historical data as possible to archival storage toward developing the capability of keeping all data forever online and accessible to the warehouse.

In a data warehouse that maintains massive quantities of history data, the volume of data is typically inversely proportional to its use. In other words, there is an enormous amount of cool history data that is accessed lightly, and a relatively lesser volume of hot and warm data that is accessed frequently.

The following, somewhat loose, default definitions apply to the commonly described temperature bands.

Temperature	Default Definition
COLD	The 20% of data that is least frequently accessed.
WARM	The remaining 60% of data that falls between the COLD and HOT bands.
HOT	The 20% of data that is most frequently accessed.
VERY HOT	Data that you or Teradata Virtual Storage think should be added to the Very Hot cache list and have its temperature set to very hot when it is loaded using the TVSTemperature query band.

Note:

Teradata Virtual Storage tracks data temperatures at the level of cylinders, not tables. Because the file system obtains its temperature information from Teradata Virtual Storage, it also handles temperature-related compression at cylinder level. For more information, see *Teradata Vantage™ - Teradata® Virtual Storage*, B035-1179.

The file system can change the compressed state of the data in an AUTOTEMP table at any time based on its temperature. Cylinders in an AUTOTEMP table become eligible for temperature-based block-level compression only when they reach or fall below the threshold defined for COLD temperature-based block level compression. See TempBLCThresh in *Teradata Vantage™ - Database Utilities*, B035-1102 for more information about the temperature settings that you can use for temperature-based block-level compression.

Temperature-based thresholds for the block-level compression of AUTOTEMP tables work as defined by the following table.

IF data blocks are initially ...	AND then become ...	THEN the file system ...
block-level compressed	warmer than the defined threshold for compression	decompresses them.
not block-level compressed	colder than the defined threshold for decompression	compresses them.

For tables that are not defined with BLOCKCOMPRESSION=AUTOTEMP, you must control their block-level compression states yourself using Ferret commands or, if a table is not populated with rows, you can use one of the TVSTemperature query bands to specify the type of block-level compression to use for the newly loaded rows. If temperature-based block-level compression is disabled but block-level compression is enabled, Vantage treats AUTOTEMP tables the same as MANUAL tables.

For all of the data in a table to be block compressed or decompressed at once in an AUTOTEMP table, Teradata Virtual Storage must become aware that all cylinders in the table have reached the threshold specified by the DBS Control parameter TempBLCThresh. This would occur in the following case. Suppose the threshold value for TempBLCThresh is set to WARM.

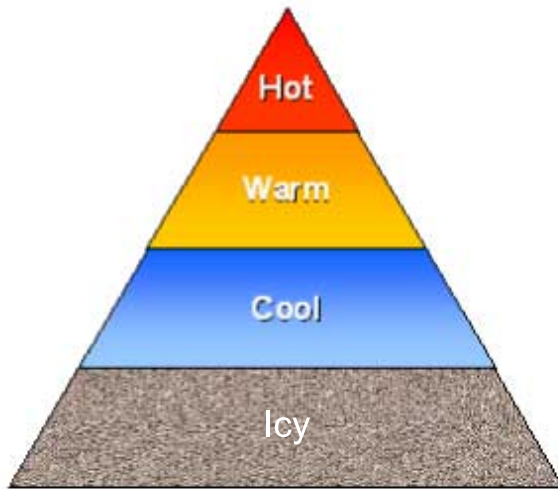
IF all of the cylinders in the table ...	THEN they all become eligible for ...
reach or fall below the WARM or COLD thresholds	block-level compression.
reach or exceed the HOT or VERY HOT thresholds	decompression.

Because of this, the best practice is not to use the AUTOTEMP option, or not to use any form of temperature-based block-level compression for a table that you think requires compression consistency for the entire table.

Extended Data Lifetimes

The lifetime of data is now being extended for a number of different reasons. Users often have varied performance requirements for time-based or historical data: recent data might be accessed frequently, while older data is accessed less often. Call these conceptual access rates hot, warm, cool, and icy, respectively, ranking from most frequently accessed to least frequently accessed. Keep in mind that these data access states are largely conceptual. Cool and icy have a loose correspondence with the temperature-based block-level compression state of COLD and the warm and hot states loosely correspond with the identically named temperature-based block-level compression states.

In a warehouse with massive historical databases, the volume of data is typically inversely proportional to the data usage, as illustrated by the following graphic, where the ordinate represents the relative warmth of the data and the abscissa represents the volume of data represented by the respective measures of data warmth.



In this picture, the temperature of the data reflects its access rate. The optimal storage for hot, warm, and cool data is online disk that is directly accessible to the data warehouse.

See the information about CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for information about how you can specify the way an individual table deals with block-level compression based on the temperature of the data.

Cool and Icy Data

Typically, there is a vast quantity of cool historical data in the data warehouse that is accessed lightly and a lesser volume of hot and warm data that is accessed frequently.

Note:

The distinction between cool and icy data is conceptual and has no direct parallel to temperature-based block-level compression.

While cool data might be accessed lightly on average, it still has temporal hot spots, such as those that occur when performing comparative analyses of sales data between different time periods. Similarly, if the relational schema is modified by adding new columns or indexes, or if column data types are changed, then the affected data cannot be considered to be dormant. Finally, if data is periodically accessed and somehow recast to make historical data relevant within the current business context, then it is not dormant.

Truly dormant data is rarely, if ever, accessed, and is typically retained for various national and international regulatory reasons such as the Sarbanes-Oxley Act in the United States, Bill 198 in Canada, or the Eighth Company Law Directive 1984/253/EEC in the European Union rather than for operational reasons. As

a result, the data stored in the warehouse is frequently a mix of both important and unimportant data (Unimportant from the perspective of the day-to-day operation of running the enterprise, not from a legal perspective), and a flexible management system is required to allocate the appropriate availability, reliability, and privacy levels for the data as its usage changes across time.

Icy, or truly dormant, data is a good candidate for alternative storage such as tape or optical disk. The treatment of icy data is not the subject of this section.

Warm, Hot, and Very Hot Data

Note:

The distinction between warm, hot, and very hot data is somewhat conceptual and is only loosely parallel to the WARM and HOT categories of temperature-based block-level compression. Very Hot data is data that you or Teradata Virtual Storage determine should be added to the Very Hot cache or that is currently contained on Very Hot cylinders.

Warm and hot data typically constitute what is often called the operational data store.

Hot and warm data are both important in the extended lifetime of data, particularly with respect to multivalue compression (see [Compression Types Supported by Vantage](#)), as is cool data to a lesser degree.

Storing Data Efficiently

Specifying DECIMAL and NUMERIC Precisions

Storage requirements for decimal data types are a function of the precision required of the values to be stored. The following table indicates the number of bytes used to store DECIMAL and NUMERIC data types of various precisions.

Number of Precision Digits	Number of Bytes Stored
1 - 2	1
3 - 4	2
5 - 9	4
10 - 18	8

You can achieve optimal space savings by using multivalue compression together with an efficient decimal size. An example would be to define decimal precision such that a 4-byte representation was used instead of an 8-byte representation, for example DECIMAL(9) instead of DECIMAL(10). Then, multivalue compression could be used on the most frequently occurring values to reduce the storage overhead still further.

Specifying INTEGER Precisions

Like the DECIMAL/NUMERIC data type, the INTEGER family of data types offers different levels of precision that you can harness to reduce the number of bytes used to store integer numbers.

The following table indicates the number of bytes used to store integer numbers of various precisions:

Integer Data Type	Number of Bytes Stored
BYTEINT	1
SMALLINT	2
INTEGER	4
BIGINT	8

You can achieve optimal space savings by using multivalued compression together with an efficient integer data type. An example would be to define the precision such that a 1-byte representation was used instead of a 4-byte representation: for example, BYTEINT instead of INTEGER. Then, multivalued compression could be used on the most frequently occurring values to reduce the storage overhead still further.

Using Standardized Encodings

There are many standardized encodings of commonly used words. For example, the United States Postal Service (USPS) encodes all U.S. states and possessions using a two-character abbreviation. The lengths of the unabbreviated state and possession strings range from four characters (Guam, Iowa, Ohio, and Utah) to 30 characters (Federated States of Micronesia). While you could define a State column as CHARACTER(30) and compress its values, or as VARCHAR(30), the easiest and most compact method of denoting U.S. states and possessions is to use the two-character USPS encodings.

Storing NUMBER Data

You can achieve optimal space savings for NUMBER data using either multivalued compression, algorithmic compression, or a combination of both.

NUMBER data is stored as a variable length field from 0 to 18 bytes in width. You can use the NUMBER type to represent both fixed point and floating point decimal numbers, depending on the syntax you use to specify the type. In the following table, the characters *p* and *s* represent precision and scale, respectively.

General Type	Syntax	Functional Description
Fixed point	NUMBER(<i>p</i> , <i>s</i>)	Similar to DECIMAL(<i>p</i> , <i>s</i>)
	NUMBER(<i>p</i>)	Similar to DECIMAL(<i>p</i>)
Floating point	NUMBER(*, <i>s</i>)	Floating point decimal with <i>s</i> fractional digits
	NUMBER	A floating point number
	NUMBER(*)	Same as NUMBER

The following table highlights the approximate equivalences/valid substitutions between the fixed and floating point NUMBER types with other exact (fixed point) and approximate (floating point) types.

You can use this NUMBER type ...	Anywhere you can use this data type ...	With a maximum precision of this many digits ...
Fixed point	<ul style="list-style-type: none"> • DECIMAL • NUMERIC 	38
Floating point	<ul style="list-style-type: none"> • FLOAT • REAL • DOUBLE PRECISION 	40

Whether a NUMBER type is used to represent a fixed point value or a floating point value, it is stored in the same way and with the same accuracy. See [Floating Point NUMBER Types](#), [Non-INTEGER Numeric Data Types](#), and *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the NUMBER type.

Storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC Data

You should consider storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data as CLOB or BLOB, or data, respectively, if the columns in question are not accessed regularly. This consideration also applies to storing VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data as XML data. This is because large objects are stored outside the base table row in subtables (subtable rows have the same row hash value as their associated base table rows, so both are stored on the same AMP), so there is no performance impact of non-LOB or non-XML queries against tables that have LOB or XML columns, while there is a performance cost to retrieving rows with inline VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC data if that data is not part of the request.

Conversely, character and byte columns that are frequently accessed should be stored as VARCHAR, VARBYTE, and VARCHAR(n) CHARACTER SET GRAPHIC if possible because of the performance cost of retrieving a LOB or XML string from its subtable.

Storing Character Data

For character data, an alternative to encodings and value compressing fixed-length CHARACTER(n) strings is to specify the variable-length VARCHAR or LONG VARCHAR data types. The number of bytes used to store each VARCHAR or LONG VARCHAR column is the length of the data item plus 2 bytes. Contrast this to the fixed-length CHARACTER data type which uses *n* bytes per row, regardless of the actual number of characters in each individual column.

The demographics of the data determine whether VARCHAR, LONG VARCHAR, or CHARACTER plus multivalue compression is more efficient. The most important factors are:

- Maximum column length
- Average column length

Evaluate the following factors when determining which approach to storing the data is the more efficient:

- VARCHAR or LONG VARCHAR are more efficient when the difference of maximum and average column length is high and value compressibility is low.

- Multivalue compression with CHARACTER data is more efficient when the difference of maximum and average column length is low and value compressibility is high.

When neither CHARACTER nor VARCHAR/LONG VARCHAR is clearly a superior choice, use VARCHAR or LONG VARCHAR because their data requires slightly less CPU resource to manipulate than CHARACTER data.

Base Table Row Format

Note:

The row format diagrams and the examples in this section portray a system with 64 KB rows. A system with larger, 1 MB rows is not shown.

The structure of base table rows is slightly different for rows in the following rectangular conditions.

- A table having an nonpartitioned primary index versus a table having a partitioned primary index.
- A system using *byte-packed* format referred to as *packed64* format versus a system using *64-bit byte-aligned* referred to as *aligned row* format.
- A table with load isolation versus a table without load isolation. Load isolation allows committed reads during table loads on tables that are created or altered to be load isolated. For more information, see *Teradata Vantage™ - Database Administration*, B035-1093.

Row format information is used to make fine-grained row size estimates for capacity planning.

General Row Structure

The following topics describe the structure of a database row in systems using packed64 format and systems using aligned row format. Whether a system stores its data in packed64 or aligned row format depends on the settings on several factors your Teradata Services personnel can modify. The size of tables on a system that stores data in packed64 format is generally between 3% and 9% smaller than the size of the same tables on a system that stores data in aligned row format (the average difference is roughly 7% smaller for the packed64 row format). Storing data in packed64 format reduces the number of I/O operations required to access and write rows in addition to saving disk space.

Either 12 or 16 bytes of the row header are devoted to overhead, depending on whether the table has a non-partitioned or a partitioned primary index and whether the row format of your system is 1 MB or 64 KB rows. Systems with 64 KB rows have 12 bytes devoted to overhead, and systems with 1 MB rows have 16 bytes devoted to overhead. The maximum row length is 1 MB. This limit is the same for both packed64 and aligned row formats. This maximum length does not include BLOB, CLOB, or XML columns, which are stored in special subtables outside the base table row. See [Sizing a LOB or XML Subtable](#) for information about BLOB, CLOB, and XML subtables.

The number of characters that can be represented by this number of bytes varies depending on whether characters are represented by one byte, two bytes, or a combination of single-byte and multibyte representations.

There are three general categories of table columns, which are stored in the row in the order listed:

1. Fixed length.

Storage is always allocated in a row for these columns.

2. Compressible.

Includes both value-compressed and algorithmically compressed data and can be stored in any order.

FOR this type of column ...	Storage is allocated ...
Multivalue-compressible	<p>in a row when needed.</p> <p>No disk storage space is required when the column contains a compressed value as specified in the CREATE TABLE DDL text.</p> <p>If a column value is not compressed, then Vantage allocates storage space for it.</p>
algorithmically-compressible	<p>in a row when needed.</p> <p>If a column is not multivalue compressed or NULL, Vantage allocates storage space for it.</p> <p>Algorithmically-compressed data requires less space.</p>

3. Variable length.

Storage space is allocated in the row depending on the size of the column.

The structure of base table rows varies slightly for tables having an nonpartitioned primary index versus a partitioned primary index (see [Row Structure for Packed64 Systems](#) and [Row Structure for Aligned Row Format Systems](#) for details).

For a system using the packed64 format, columns are stored in field ID order, with the fixed length fields first followed by the compressed fields, and last by the variable length fields.

In aligned row format, fixed length columns are stored first, followed by compressed fields, and then by variable length fields. This ordering is the same as the ordering for packed64 format storage with the exception that the columns within each category are stored in decreasing alignment constraint order.

For example, if a row contains fixed length columns with the types CHARACTER, INTEGER and FLOAT, the floating point numbers are stored first, followed by the INTEGER numbers, and then by the CHARACTER columns regardless of the order in which the columns were defined in the CREATE TABLE request defining the table.

Vantage uses a space optimization routine for aligned format rows that can store a maximum of 7 bytes of data in the potentially unused space that can occur when a column is aligned on a 0(mod 8) boundary. After the routine fills the available space with candidate data, Vantage follows the rules outlined in the preceding paragraphs to complete the remainder of the row data in an aligned format row.

Sector Alignment

The Teradata File System supports devices that use either a native 512 byte sector size or a native 4KB sector size.

The new 4KB disk drives can be separated into 2 classes: those that support I/O on non-4KB aligned disk boundaries and those that do not support non-4KB aligned I/O. When operating on a system with 4KB drives, Vantage only performs I/Os on blocks that are full 4KB aligned (in size and length), regardless of the class of the drive.

Vantage only supports a homogeneous alignment configuration throughout the entire system. On a given system, data on all storage devices is aligned or unaligned, and cannot be both. This also applies across cliques.

General Row Structure When Compressing Variable Length Columns

The following information describes aspects of general row structure when one or more variable length columns in a table are compressed. The description applies to columns having any of the following data types:

- VARBYTE
- VARCHAR
- VARCHAR(n) CHARACTER SET GRAPHIC
- UDT
- CLOB/BLOB

The description assumes that all variable length compressible columns are treated as an extension of fixed length compressible columns except that decompressed variable length columns store both a length and the actual column data. There is an additional presence bit for an algorithmically compressed column to indicate whether the column data is compressed or not. Vantage sets this bit only when data in a column is compressed using algorithmic compression.

When column data is compressed algorithmically, Vantage stores it as length: data pairs interleaved with the other compressible columns in the table. When column data is null, Vantage does not store a length, so there is no overhead in that case.

The following summary information applies to general row structure elements for the storage of variable length column data for multivalue compression.

- When compression does not apply to a value in a column that specifies multivalue compression, Vantage stores the column data in the row as a length and data value pair.

If the length of the column data is ≤ 255 bytes, then Vantage stores the length in 1 byte; otherwise it stores the length in 2 bytes.

A variable length column (without compression) always has a 2-byte offset associated with it, whereas a compressed variable length column can have its length stored in one or two bytes, depending on whether the column length is ≤ 255 or not, respectively.

- Vantage stores uncompressed variable length data in the row as a length and data value pair.

The stored length for multivalue compressed data is the length of the uncompressed column value.

- If a variable or fixed length multivalue compressed column is compressed or is null, then its values are not stored in the row.

Instead, Vantage stores one instance of each compressed value within a column in the table header and references it using the presence bits array for the row.

- If a fixed length multivalue compressed column is not compressed, then Vantage stores its values in the row as data, but without an accompanying length. No length is needed because Vantage pads fixed length data values to the maximum length defined for their containing column.

The following summary information applies to general row structure elements for the storage of variable length column data for algorithmic compression.

- If a variable length column is not compressed, then Vantage stores its values in the row as a length and data value pair.
- If a fixed length column is not compressed, then Vantage stores its values in the row as data, but without an accompanying length.

Vantage pads fixed length data values to the maximum length defined for their containing column.

- For an algorithmically-compressed column with a variable length, Vantage stores its values in the row as a length and data value pair.

The stored length for algorithmically-compressed data is the length of the compressed column value, not its original, uncompressed length.

If the length of the column data is ≤ 255 bytes, then Vantage stores the length in 1 byte; otherwise, it stores the length in 2 bytes.

- For an algorithmically-compressed column with either a variable length or a fixed length, Vantage does not store a representation of its data in the row if the column is null, but does indicate its existence in the data using the presence bit array for the row.

The following sets of examples demonstrate the specific effects on row structure for multivalue compression alone (cases 1, 2, and 3 of example 1), algorithmic compression alone (cases 1, 2, and 3 of example 2), and combined multivalue and algorithmic compression (cases 1, 2, and 3 of example 3). In each case, the row structure diagram is based on the structure for a partitioned table (see [Packed64 Row Structure for a Partitioned Table](#)) even though the examples are all for nonpartitioned primary index tables.

Example 1: Algorithmic Compression But No Multivalue Compression

Suppose you define the following table to use Huffman encoding algorithms to compress and decompress two of its columns algorithmically. This table specifies algorithmic compression on columns *alc1* and *alc2*, but no multivalue compression. Particularly relevant data for each case is highlighted in **boldface type**.

```
CREATE TABLE t1 (
  fc1  INTEGER,
  alc1 VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
                                ALGDECOMPRESS huffdecomp,
  vc1  VARCHAR(20),
  alc2 VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
                                ALGDECOMPRESS huffdecomp,
  vc2  VARCHAR(20) );
```

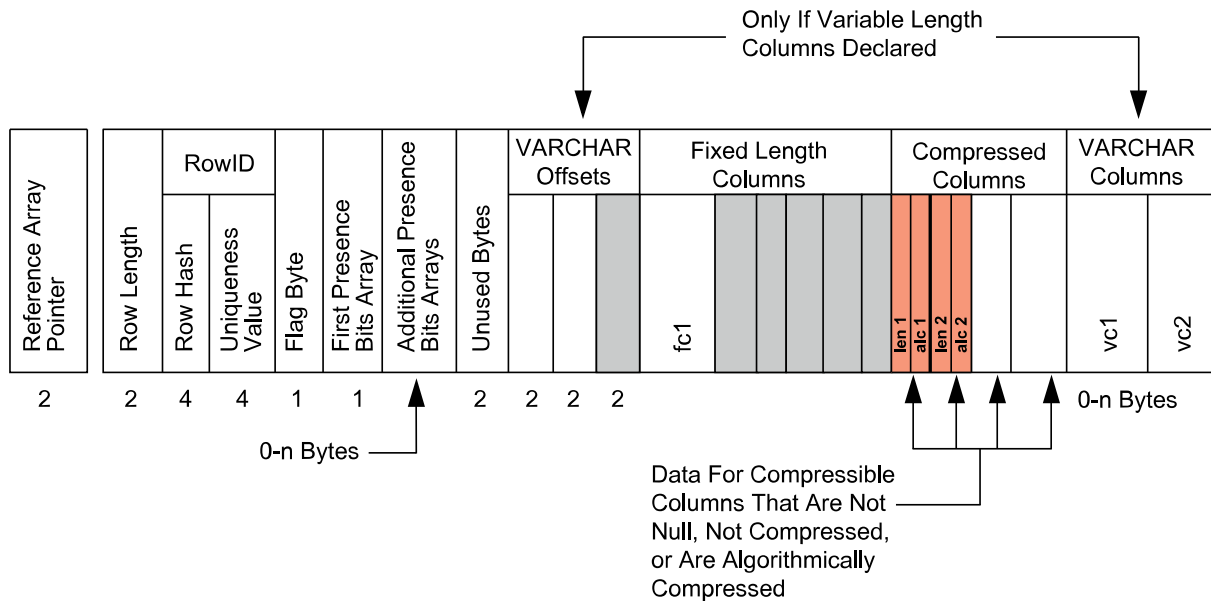
Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Off- set	Num- Comp- ress	Next Fld Ind	UDT or Com- press offset	Calc offset	Sto- rage	Pres- ence Byte	Bit	Sort Desc	EvlRepr	
1	1025	20		2	0	Offset	Nullable	0	1	AscKey	DBC	INTEGER
2	1026	-		4	196	Comprs	alc+Null1	0	2	NonKey	DBC	VARCHAR(10) LATIN
3	1027	12		5	0	Var	Nullable	0	4	NonKey	DBC	VARCHAR(20) LATIN
4	1028	-		3	206	Comprs	alc+Null1	0	5	NonKey	DBC	VARCHAR(10) LATIN
5	1029	14		0	0	Var	Nullable	0	7	NonKey	DBC	VARCHAR(20) LATIN

To be as general as possible, each row structure diagram indicates the configuration of fields as they would be if the table had a partitioned primary index, though none of the actual table creation SQL text specifies a PPI. The diagrams also assume the system has a Packed64 row structure.

Case 1

Suppose you have the same table, but without any algorithmic compression being defined. The row structure for this table looks something like the following diagram.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a <i>variable length data type</i> .
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a <i>variable length data type</i> .

Abbreviation	Definition
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

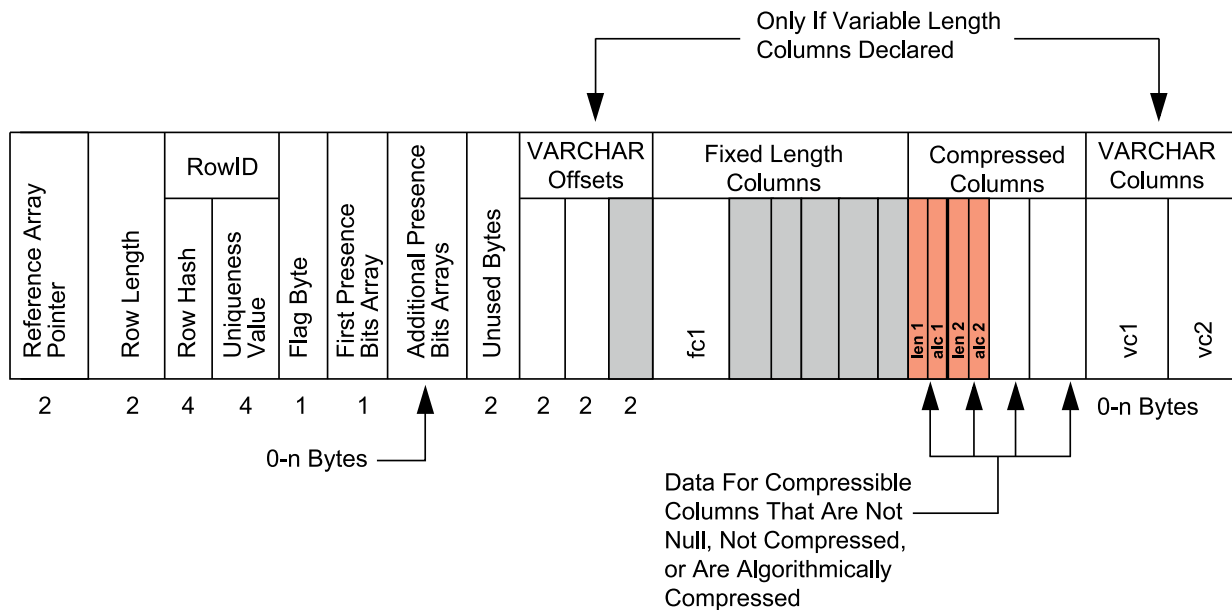
In this case, the data values for columns *alc1* and *alc2* are both stored in the row and are not compressed.

Vantage has placed the variable length compressible columns after the fixed length columns in the compressible columns area of the row. This is also reflected in the Field5 descriptor. Because the columns are not compressed, Vantage stores lengths with the two *alc* columns. *len1* and *len2* contain the uncompressed lengths of those columns, and the ALC bit is not set.

Case 2

In the next case, algorithmic compression has been defined, and the data for columns *alc1* and *alc2* is not null. *alc1* and *alc2* are present in the row and compressed. *len1* and *len2* contain the new compressed length. The ALC bit has been set for this case to indicate that the data in columns *alc1* and *alc2* is compressed.

For this case, the row structure for the table superficially looks exactly like the row structure diagram for the previous case.



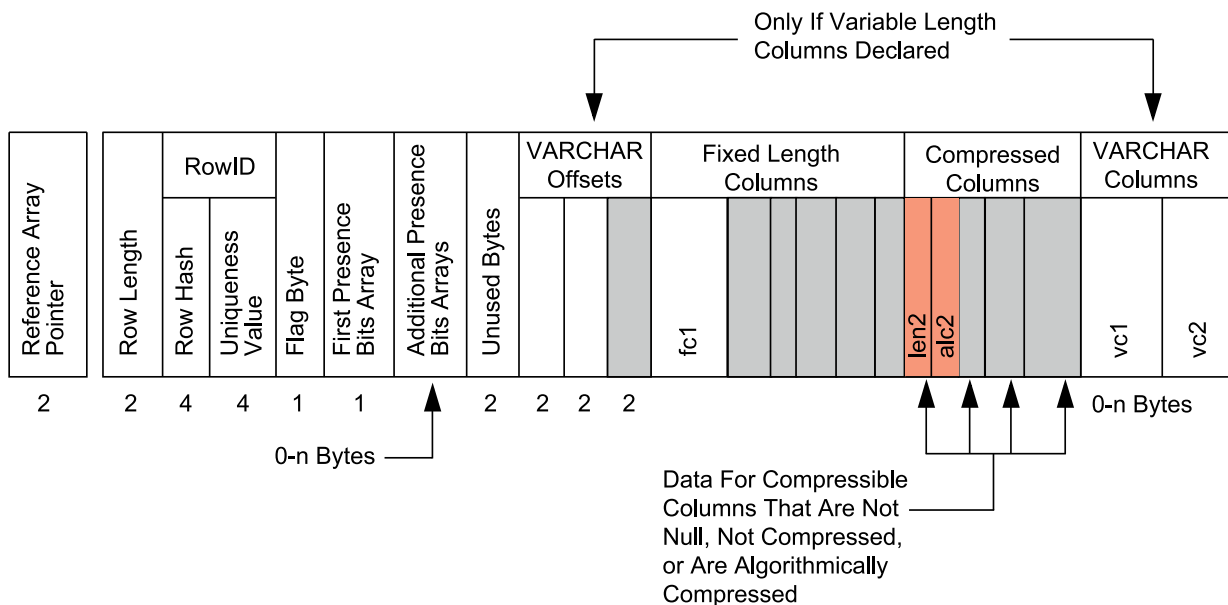
The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a variable length data type.

Abbreviation	Definition
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a <i>variable length data type</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

Case 3

In the next case, algorithmic compression has been defined, but the data for columns *alc1* and *alc2* is null. *alc1* and *alc2* are present in the row and compressed. *len1* and *len2* contain the new compressed length. The ALC bit is not set for this case because column *alc1* is null.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no length for column <i>alc2</i> is stored in the row.
alc1	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
len2	Length of the data in column <i>alc2</i> , an algorithmically-compressed variable length data type.
alc2	Algorithmically-compressed data for column <i>alc2</i> , a variable length data type.

Abbreviation	Definition
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

Example 2: Multivalue Compression But No Algorithmic Compression

This example presents row structure cases based on the following table definition. This table has multivalue compression defined on columns *mvc1* and *mvc2*, but no algorithmic compression. Particularly relevant data for each case is highlighted in **boldface type**.

```
CREATE TABLE t1 (
  fc1 INTEGER,
  mvc1 VARCHAR(10) COMPRESS ('mars','saturn','jupiter'),
  vc1 VARCHAR(20),
  mvc2 VARCHAR(10) COMPRESS ('Germany','France','England'),
  vc2 VARCHAR(20) );
```

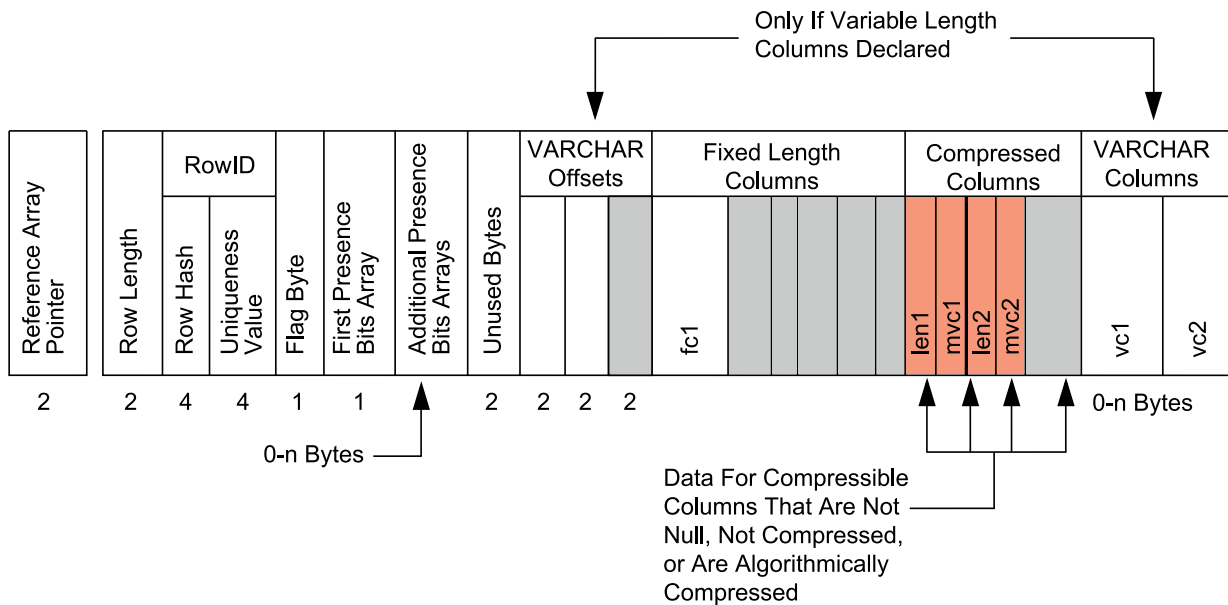
Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Off- set	Num- Comp- ress	Next Fld Ind	UDT or Com- press offset	Calc offset	Sto- rage	Pres- ence Byte	Bit	Sort Desc	Ev1Repr
1	1025	20		2	0	Offset	Nullable	0	1	AscKey	DBC INTEGER
2	1026	-		4	196	Comprs	cmp+Null12	0	2	NonKey	DBC VARCHAR(10) LATIN
3	1027	14		5	0	Var	Nullable	0	5	NonKey	DBC VARCHAR(20) LATIN
4	1028	-		3	206	Comprs	cmp+Null11	0	6	NonKey	DBC VARCHAR(10) LATIN
5	1029	16		0	0	Var	Nullable	0	1	NonKey	DBC VARCHAR(20) LATIN

Case 1

In this case, the data values for columns *mvc1* and *mvc2* are both stored in the row because they are not compressed for the particular data values they contain.

Vantage has placed the variable length compressible columns after the fixed length columns in the compressible columns area of the row. This is also reflected in the Field5 descriptor. *len1* and *len2* contain the uncompressed lengths of those columns.

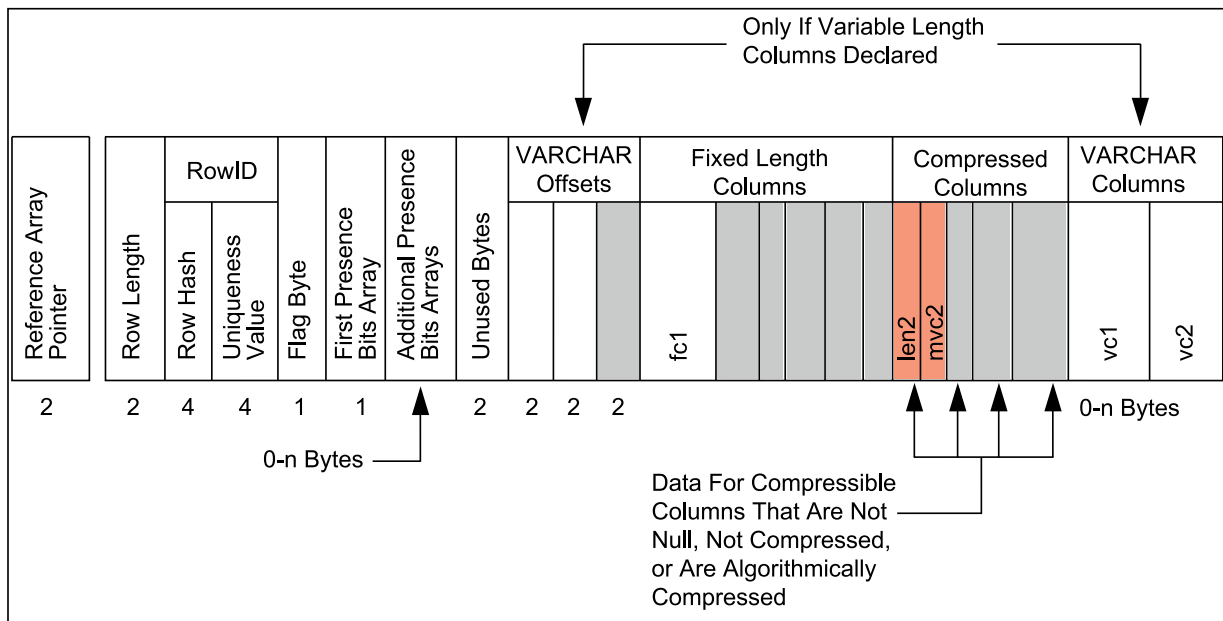


The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Compressed length of the data in column <i>mvc1</i> , a multivalue compressed variable length column.
mvc1	Multivalue compressed data for column <i>mvc1</i> .
len2	Compressed length of the data in column <i>mvc2</i> , a multivalue-compressed variable length column.
mvc2	Multivalue compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

Case 2

The next case demonstrates row storage when the column data in the row is to be multivalue compressed. In this case, the value for column *mvc1* is not stored in the row because it is compressed. The value is instead stored in the table header and referenced by the presence bits array for the row.

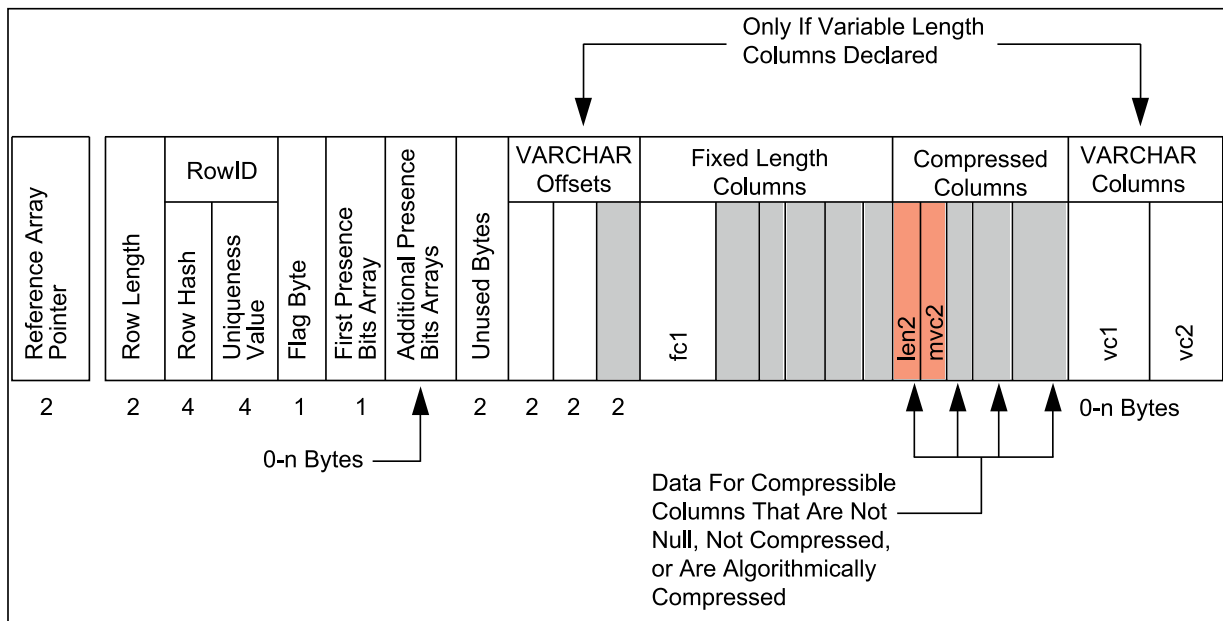


The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no length for column <i>mvc1</i> is stored in the row.
mvc1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
len2	Compressed length of the data in column <i>mvc2</i> , a multivalue-compressed variable-length column.
mvc2	Multivalue compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , an uncompressed variable-length data type.

Case 3

The next case demonstrates row storage when the column data for *mvc1* is null. In this case, the value for column *mvc1* is not stored in the row because nulls are always compressed. This demonstrates that nulls never incur a cost overhead for compression.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed-length data type.
len1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
mvc1	Not represented. This column is null, so no value for <i>mvc1</i> is stored in the row.
len2	Compressed length of the data in column <i>mvc2</i> , a multivalue compressed variable-length column.
mvc2	Multivalue compressed data for column <i>mvc2</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable-length data type.
vc2	Uncompressed data for column <i>vc2</i> , a variable-length data type.

Example 3: Mix of Multivalue and Algorithmic Compression

This example presents row structure cases based on the following table definition. This table has multivalue compression defined on columns *mvc_f1* and *mvc_v1*, and algorithmic compression defined on column *alc1*. Particularly relevant data for each case is highlighted in **boldface type**.

```
CREATE TABLE t1(
  fc1    INTEGER,
  alc1   VARCHAR(10) COMPRESS ALGCOMPRESS huffcomp
```

```

                                ALGDECOMPRESS huffdecomp,
vc1    VARCHAR(20),
mvc_f1 CHARACTER(10) COMPRESS ('Germany','France','England'),
mvc_v1 VARCHAR(20) COMPRESS ('Nike','Reebok','Adidas') );

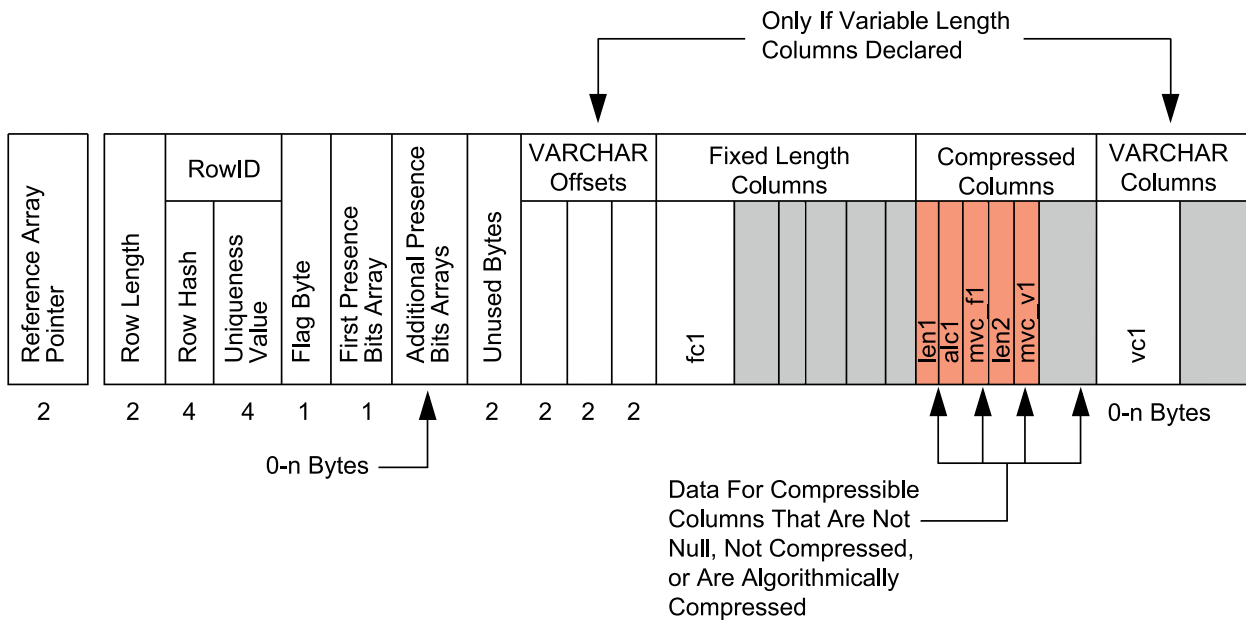
```

Field 5 of the table header for this table has the following field descriptors.

In- dex	Fld id	Off- set	Num- Comp- ress	Next Fld Ind	UDT or Com- press offset	Calc offset	Sto- rage	Pres- ence Byte	Bit	Sort Desc	Ev1Repr
1	1025	20		2	0	Offset	Nullable	0	1	AscKey	DBC INTEGER
2	1026	-		4	196	Comprs	alc+Null1	0	2	NonKey	DBC VARCHAR(10) LATIN
3	1027	14		0	0	Var	Nullable	0	5	NonKey	DBC VARCHAR(20) LATIN
4	1028	-		5	226	Comprs	cmp+Null2	0	6	NonKey	DBC VARCHAR(10) LATIN
5	1029	-		3	256	Comprs	cmp+Null2	1	1	NonKey	DBC VARCHAR(20) LATIN

Case 1

In this case, no compression has been specified for the particular data values, so data for columns *alc1*, *mvc_f1*, and *mvc_v1* is stored in the row.



The following abbreviations are used for the various fields of the diagram for this case.

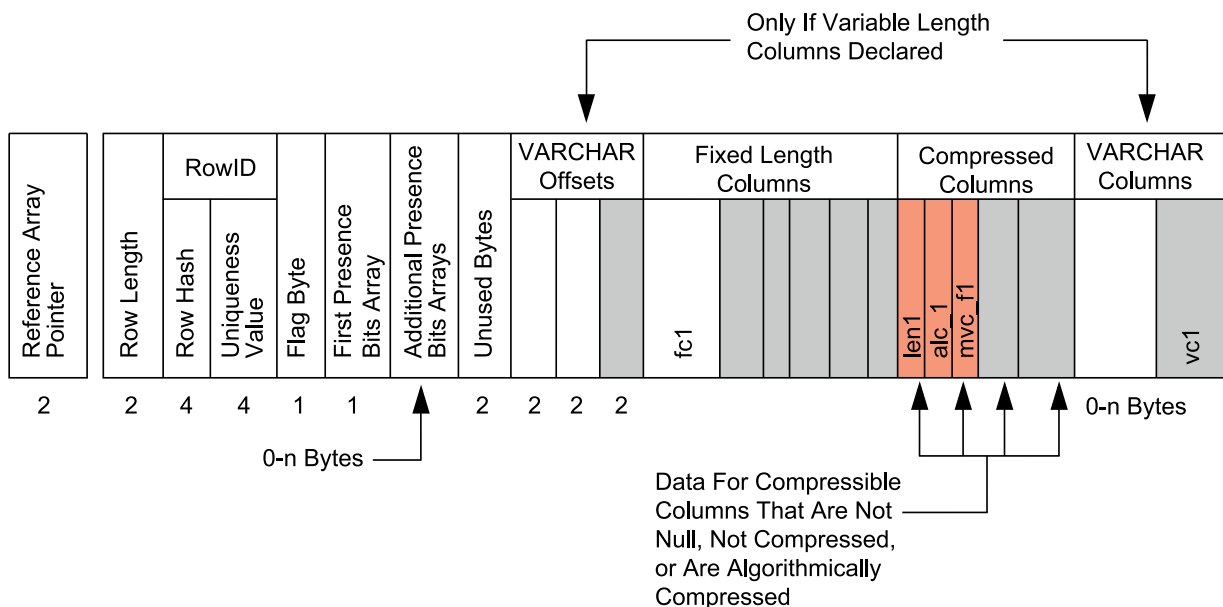
Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Compressed length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.

Abbreviation	Definition
alc1	Algorithmically-compressed data for the variable length column <i>alc1</i> .
mvc_f1	Multivalued compressed data for the fixed length column <i>mvc_f1</i> .
len2	Length of the data in column <i>mvc_f1</i> , a multivalued compressed variable length data type.
mvc_v1	Multivalued compressed data for the variable length column <i>mvc_v1</i> .
vc1	Uncompressed data for column <i>vc1</i> , a variable length data type.

Vantage has placed the variable compressible columns after the fixed field columns in the compressible columns area. This is also reflected in the Field5 descriptor. The compressed column data is interleaved with other compressible fields. *len1* and *len2* contain the uncompressed lengths of *alc1* and *mvc_v1*, respectively.

Case 2

In this case, Vantage compresses the data for columns *mvc_v1* and *alc1* because the values for the row are specified in the multivalued for those columns.



The following abbreviations are used for the various fields of the diagram for this case.

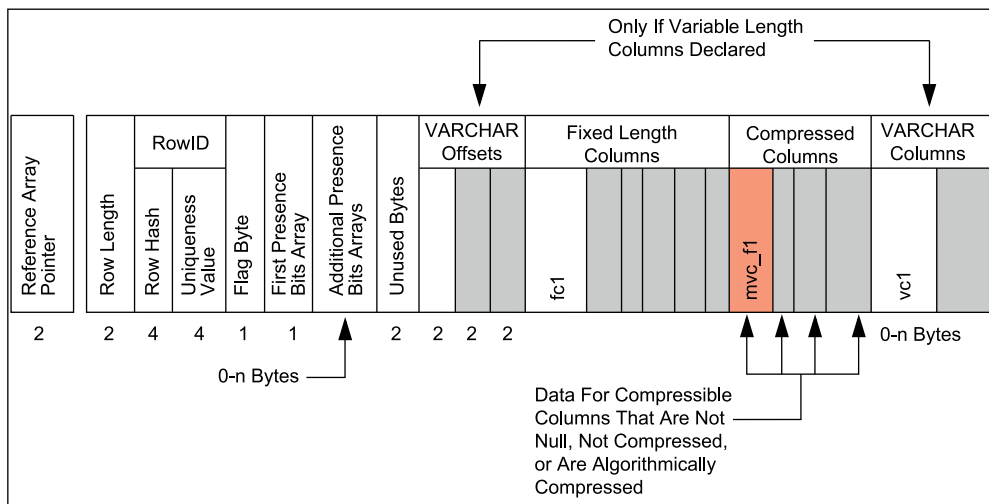
Abbreviation	Definition
fc1	Uncompressed data for column <i>fc1</i> , a fixed length data type.
len1	Compressed length of the data in column <i>alc1</i> , an algorithmically-compressed variable length data type.
alc1	Algorithmically-compressed data for column <i>alc1</i> , a variable length data type.

Abbreviation	Definition
<i>mvc_f1</i>	Multivalued compressed data for column <i>mvc_f1</i> , a fixed length data type.
<i>mvc_v1</i>	Not represented. This column is compressed, so no value for <i>mvc1</i> is stored in the row.
<i>vc1</i>	Uncompressed data for column <i>vc1</i> , a variable length data type.

In this case, Vantage does not store the value for *mvc_v1* in the row because it is compressed for the particular data value, so the value is stored in the table header. *len1* contains the compressed length for *alc1*.

Case 3

In this case, the data for the compressible columns *mvc_v1* and *alc1* is null, so Vantage does not store values for those columns in the row. This example demonstrates that nulls never incur a cost overhead for compression.



The following abbreviations are used for the various fields of the diagram for this case.

Abbreviation	Definition
<i>fc1</i>	Uncompressed data for column <i>fc1</i> , a fixed length data type.
<i>len1</i>	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
<i>alc1</i>	Not represented. This column is null, so no value for <i>alc1</i> is stored in the row.
<i>mvc_f1</i>	Multivalued compressed data for column <i>mvc_f1</i> , a fixed length data type.
<i>len2</i>	Not represented. This column is null, so no value for <i>mvc_v1</i> is stored in the row.

Abbreviation	Definition
mvc_v1	Not represented. This column is null, so no value for <i>mvc_v1</i> is stored in the row.
vc1	Uncompressed data for column <i>vc1</i> , a variable length data type.

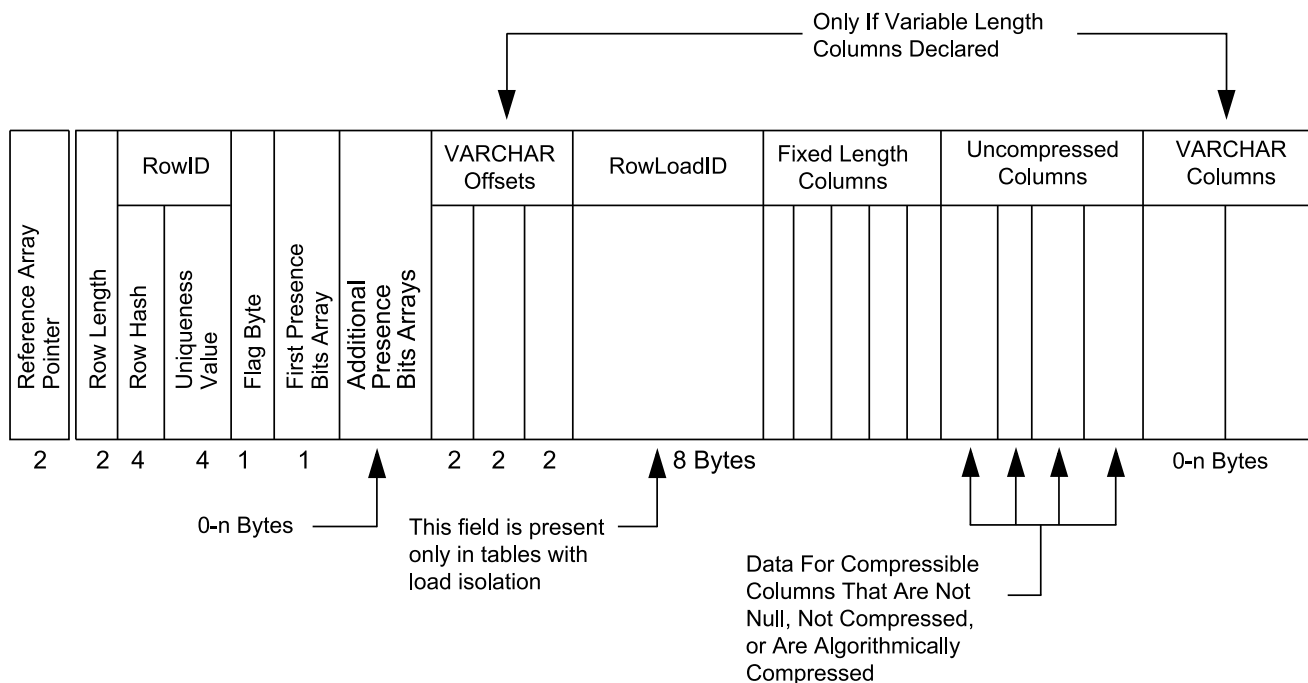
Row Structure for Packed64 Systems

Base table rows are stored in packed format on packed64 format systems, so they need not align on 8-byte boundaries. Because of this, their row structure is simpler than that of equivalent base table rows on aligned row format systems (see *Row Structure for Aligned Row Format Systems*) below.

Note that the Row Hash value is 4 bytes wide irrespective of the number of hash buckets the system has.

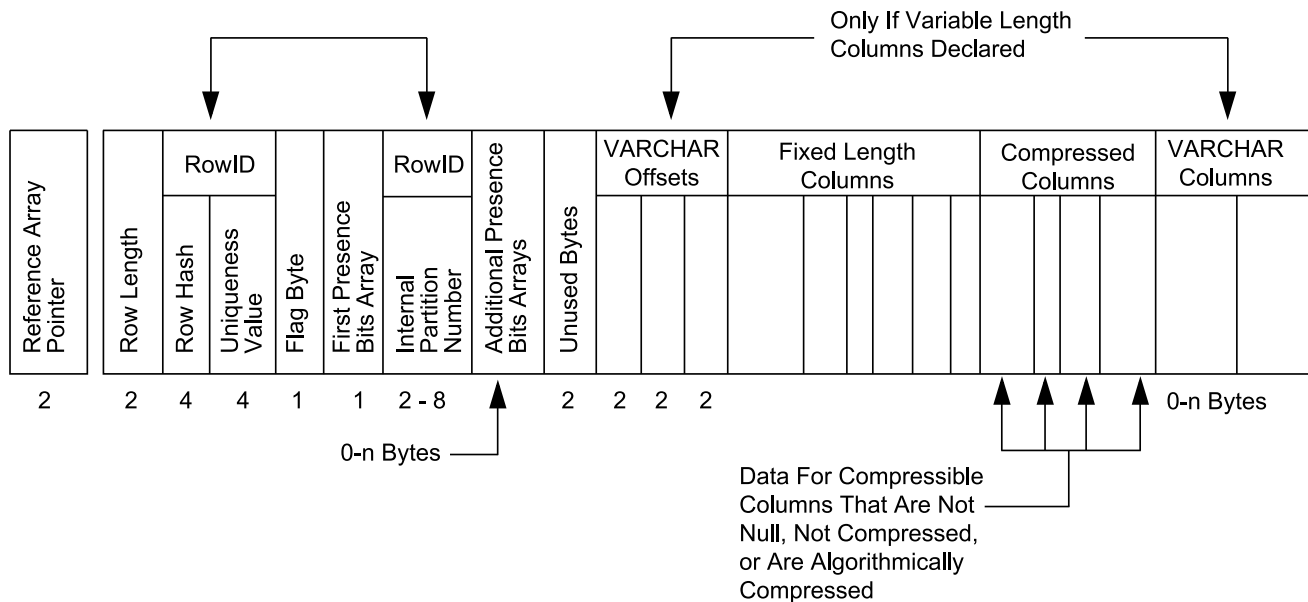
Packed64 Row Structure for a Nonpartitioned Primary Index Table with Load Isolation

The following graphic illustrates the basic structure of a database row from a table on a Packed64 format system with an nonpartitioned, or traditional, primary index. The table is load isolated, so the RowLoadID field is present.



Packed64 Row Structure for a Partitioned Table

The following graphic illustrates the basic structure of a database row from a table on a packed64 format system with a partitioned primary index:



The difference between this and the format of a nonpartitioned primary index row is the presence of an additional 2-byte or 8-byte partition number field, which is also a component of the RowID (partitioned table rows are an additional 4 bytes wider if they also specify multivalue compression). It is this field that generates the need for a `BYTE(10)` data type specification for a RowID. For nonpartitioned primary index tables, the partition number is assumed to be 0, so the rowID of a nonpartitioned primary index table is also logically `BYTE(10)` (see [ROWID Columns](#)).

Row Structure for Aligned Row Format Systems

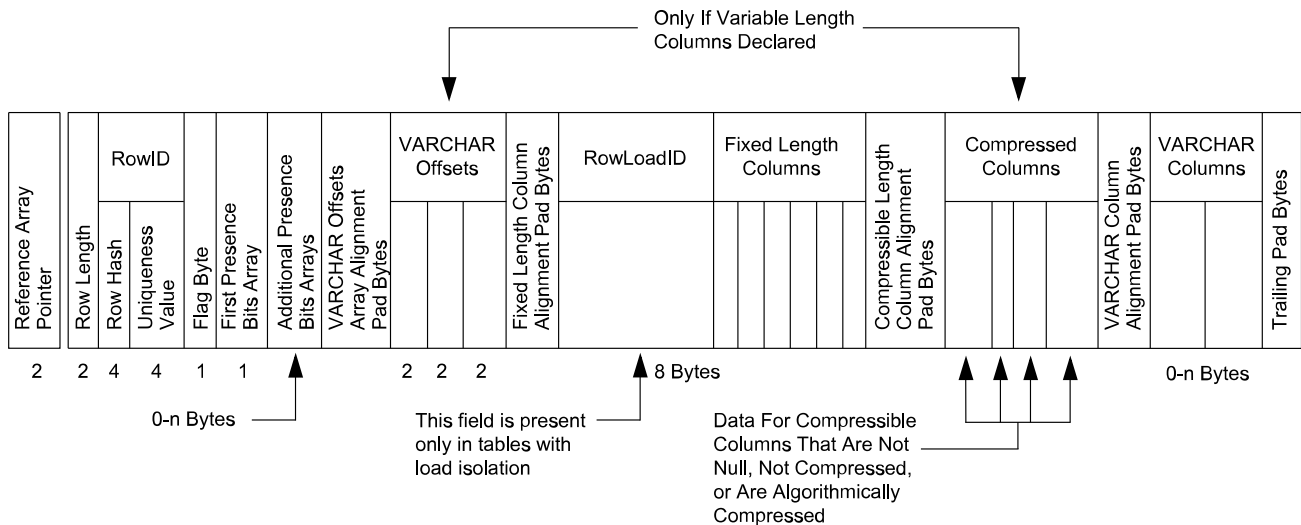
The row structure diagrams for aligned row format systems differs from those of packed64 systems by having five additional pad byte fields to ensure row alignment on an 8-byte boundary. The following table lists each of these pad fields and explains their purpose:

Pad Byte Field Name	Purpose
VARCHAR Offsets Array Alignment Pad Bytes	Aligns VARCHAR offsets array at a 2-byte boundary.
Fixed Length Column Alignment Pad Bytes	Aligns fixed length columns.
Compressible Length Column Alignment Pad Bytes	Aligns value compressible length columns.
VARCHAR Column Alignment Pad Bytes	Aligns variable length columns.
Trailing Pad Bytes	Aligns entire row on an 8-byte boundary.

Note that the Row Hash value is 4 bytes wide irrespective of the number of hash buckets the system has.

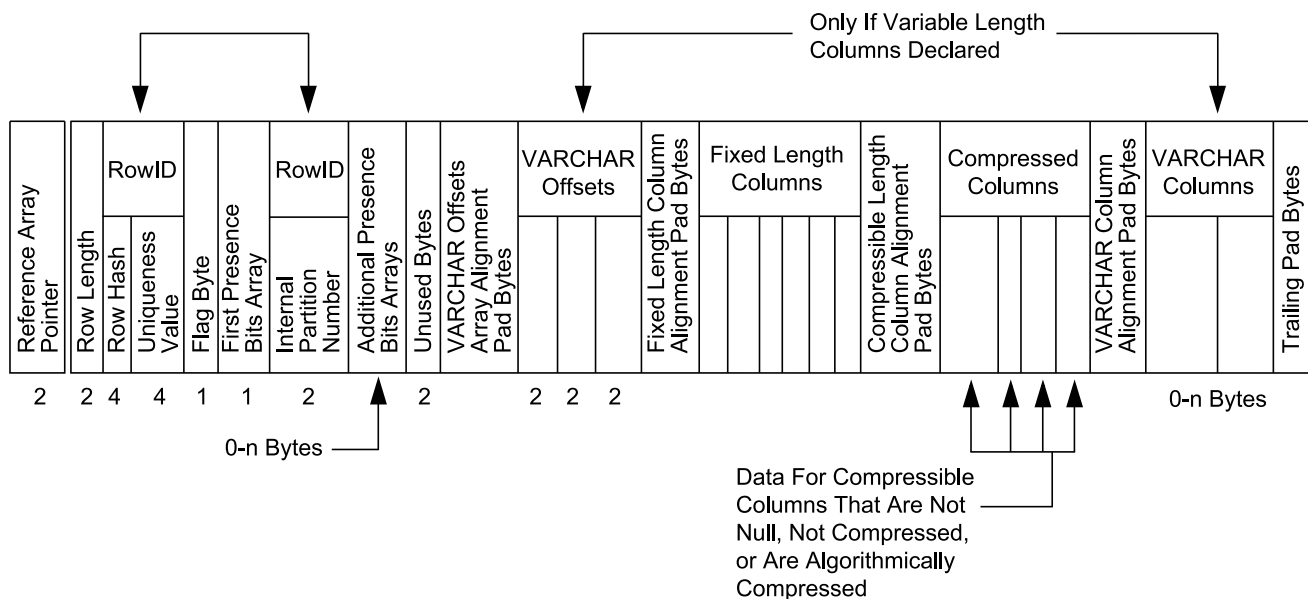
Aligned Row Structure for a Nonpartitioned Table with Load Isolation

The following graphic illustrates the basic structure of a database row from a table on an aligned row format system with a nonpartitioned, or traditional, primary index. The table is load isolated, so it has the RowLoadID field.



Aligned Row Structure for a Partitioned Table With 65,535 or Fewer Combined Partitions

The following graphic illustrates the basic structure of a database row from a table on an aligned row format system with a partitioned primary index that has 65,535 or fewer combined partitions:



The difference between this and the format of a nonpartitioned table row is the presence of a 2-byte or 8-byte partition number field, which is also a component of the RowID (partitioned table rows are an additional 4 bytes wider if they also specify multivalue compression. It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index tables, the partition number is assumed to be 0, so the rowID of a nonpartitioned primary index table is also logically BYTE(10) (see [ROWID Columns](#)).

Containers and Subrows

The data in a column-partitioned table can be stored in containers or subrows. The following table defines these terms.

Containers and Space

If Vantage can pack many column partition values into a container, this form of compression, called *row header compression*, can reduce the space needed for a column-partitioned table or join index compared to the same object without column partitioning.

If Vantage can place only a few column partition values in a container because of their width, there can actually be a small increase in the space needed for a column-partitioned table or join index compared to the same object without column partitioning. In this case, ROW format may be more appropriate.

If there are only a few column partition values (because it is possible with row partitioning that only a few column partition values occur for each combined partition) and there are many column partitions, there can be a very large increase in the space needed for a column-partitioned object compared to the same object without column partitioning. In the worst case, the space required for a table can increase by nearly 24 times.

In this case, consider making one of the following changes:

- Alter the column or row partitioning to allow for more column partition values per combined partition.
- Remove the column partitioning from the table or join index.

Container Contents

A container must have the same internal partition number and hash bucket for all the column partition values in that container row.

Following are the contents of a container:

- The row header for a container indicates its internal partition number, hash bucket, and uniqueness value for the first column partition value. The row header is the same as for any other physical row, including physical row length, a rowID, flag byte, and first presence byte. The row header for a container is either 14 or 20 bytes long.

There is only one row header for a container, using the rowID of the first column partition value as the rowID of the container instead of there being a row header for each column partition value as is the case for all other row types in Vantage. You can determine the rowID of a column partition value by its position within the container.

Note that the first presence byte for a container is not used as a presence byte.

- The first presence byte for a container is not used as a presence byte. It indicates whether autocompression types have been determined for the container.

If the selected autocompression types, which might include applying user-specified compression, do not reduce the size of the container row, the AC bit in the first presence byte in the row header is set to 0 and the container row is not autocompressed.

- The number of column partition values, including values for logically deleted rows, is represented by the container and various offsets to its sections.
- An optional series of column partition values for the local value list compression dictionary.

This is preceded by arrays of offsets if the values are variable length.

- A series of fixed-length column partition values or a series of variable-length column partition values, each prefixed by a 1-byte or 2-byte length.

The series can be empty if the autocompression bits do not indicate that any column partition values are present. If this occurs, the column partition values have all been compressed.

- 0 or more bytes of free space.
- Optional autocompression presence bits, value length compression bits, algorithmic compression bits, and run length bits, depending on the autocompression techniques used in the container row in reverse order of the series of values.

A column container should have thousands of column values for fixed length and short variable-length data types unless the table is overly row-partitioned. The values and presence bits grow closer to one another as they consume the available free space. If there is insufficient free space, Vantage expands the row.

For a single-column partition that has COLUMN format (that is, its physical rows are containers), a column partition value is the same as a column value and can have either a fixed or a variable length.

For a multicolumn partition with COLUMN format, a column partition value has a structure similar to a regular row, containing presence and compression bits as 0 or more bytes, offsets to variable-length column values, the values of its fixed-length columns, the values of its uncompressed columns, and the values of its variable-length columns.

A container does not include:

- A row length because the length of a column partition value is handled separately
- An internal partition number
- A hash bucket
- A uniqueness value
- Presence or compression bits
- First presence bit

Vantage applies user-specified compression within the multicolumn column partition value and might apply autocompression to the column partition value as a whole.

Column partition values for a single-column or multicolumn column partition can have either fixed or variable length. If the column partition value has variable length, the length is not part of the column partition value.

Instead, the length is specified in the container row by a preceding length field or by using the difference between offsets if the column partition value is in the local value-list dictionary.

A container includes other information such as offsets to the beginning of the series of column partition values.

Containers and Autocompression

A container with autocompression includes:

- 2 bytes are used as an offset to the compression bits.
- 1 or more bytes indicate the autocompression types and their arguments for the container.
- 1 or more bytes of autocompression bits, depending on the number of column partition values and the autocompression type.
- 0 or more bytes are used for a local value-list dictionary.
- 0 or more bytes are used for present column partition values.

A container without autocompression uses 0 or more bytes for present column partition values. A container can exist without autocompression because:

- You specified NO AUTO COMPRESS for the column partition when you created the table or join index.
- No autocompression types are applicable for the column partition values of the container.

Row Structure for Containers (COLUMN Format)

Vantage packs column partition values into a container up to a system-determined limit and then packs the next set of column partition values into a new container. The column partition values within a container must be in the same combined partition to be packed into that container.

A column partition represents one or more table columns of a table. A column partition that has COLUMN format is represented as a series of containers that hold the column partition values of the column partition.

A container consists of a header and column partition values followed by autocompression bits at the end, with free space in between. The free space is allocated in such a way that column partition values and autocompression bits can grow toward each other using the free space without moving around the values and changing the row size.

A newly constructed container in memory starts at the maximum allowed size and, therefore, a large free space. Once it either fills up or there are no more column partition values to add for the current DML request, the container can be reduced in size as described later, and it is then written to disk. If there are still more column partition values to add for this request, Vantage starts a new container in memory.

When new column partition values arrive to be appended for a subsequent DML request and there is a last container for the combined partition in which the column partition values are to be appended, and subsequently insufficient free space is available for appending the next new column partition value in this last container, and the container was reduced in size when last written as described later, Vantage expands the container in memory to its maximum allowable size if that would enable a column partition value to be added. If the container becomes full, it can be reduced in size as described later, and it is then written to disk. The process begins again with a new container.

Before writing a container that has reached its maximum size, either because it is a new container or because it is a last container read from disk that was expanded to the maximum size because it ran out free space, when there are no more column partition values to be added to it by a DML request, there is sufficient free space to add more column partition values, but its free space exceeds a system-determined percentage of its size without the free space, Vantage reduces its free space to be within this percentage.

It is possible that free space could occur in the last container for each combined partition. If there are many combined partitions, the sum of the free space could add up to a great deal of unused disk space. Therefore, it is desirable to keep this total unused space to a small percentage of the table or join index size. However, having a reasonable amount of free space in the last container minimizes the copying of a container to a larger memory area in order to accommodate new incoming column partition values such as, for example, by an array INSERT, a small INSERT SELECT request, or a large INSERT SELECT request to a row partitioned column-partitioned table or join index such that a small number of column partition values are inserted into a combined partition at a time, and most often the container can be written with the same physical row size after the insert operation, which is more efficient than if the physical row size changes.

When writing a container at the point where it can no longer hold additional column partition values, necessitating that a new container be started, the free space for the container is reduced to 0 or near zero. The last container does not necessarily need to be written if it were just read and there is insufficient free space to add another column partition value but the remaining free space is small. However, if a container has too much remaining free space, Vantage must remove the free space, and the last container row must be written.

This does not apply to a container for the delete column partition and is autocompressed.

Container for the Delete Column Partition

A container for the delete column partition has fixed-length, single-column, BYTEINT NOT NULL column partition values. This data is constrained to have the values 0 or 1. The container row has the layout as a fixed-length container.

Row Structure for Subrows (ROW Format)

A column partition that has ROW format is represented as a series of subrows, where each subrow contains a single column partition value of the column partition. Subrows have the same format as regular rows with the following exceptions:

- A regular row contains all the column values of a table row, while a subrow contains only a subset of the column values of a table row (that is, a column value for each of the columns in the column partition).
- The internal partition number of the row ID of a regular row does not indicate a column partition because regular rows are not column-partitioned.

The internal partition number of the row ID of a subrow indicates its column partition number.

Vantage applies any user-specified compression within the subrow for the column partition value.

Alignment of Containers and Subrows

Containers and subrows are always packed, even on aligned format systems. The only differences between packed and aligned systems are:

- The length of a container or subrow is a multiple of 8 on 64-bit aligned system.
- The length of a container or subrow is a multiple of 2 on a packed system.

For a container, the file system adds any extra bytes that are required to make the length of the row even to the freespace, not to the end of the row.

On a packed system, the length of a subrow can be odd. If so, the file system adds a byte to the end of the subrow.

Column Partitioning

Column partitioning is a form of partitioning for multiset tables and for single-table, non-aggregate, non-compressed join indexes. Columnar storage stores the data into a series of containers with usually many values of the column partition packed into each container; alternatively, the values of a column partition can be stored into a series of subrows with one value of the column partition per subrow.

Column partitioning enables sets of table or join index columns to be stored in separate partitions. Row partitioning of primary-indexed tables also enables sets of rows to be stored in separate partitions. Teradata Columnar makes it possible for a table or join index to be column-partitioned, or both column-partitioned and row-partitioned by using multilevel partitioning.

Column partitioning enables the Optimizer to devise efficient searches by using column and row partition elimination based on the columns that are needed by a query. If a table or index column is not needed by a request, the column partition with that column need not be read. If multiple columns are needed for a request, the query plan devised by the Optimizer includes putting projected column values from selected table rows together to form result rows. This can be combined with row partition elimination to further reduce the data that must be accessed to satisfy a request.

Vantage can apply various compression techniques to column-partitioned data that can reduce the storage requirements for a table or join index, which can then reduce the I/O requirements for DML requests. When column partitioning is combined with row partitioning, the number of compression opportunities available to the system can increase.

Consumption of Disk Space by Populated and Empty Partitions

With the large number of partitions that can be defined for a table or join index, it is very likely that a high percentage of those partitions are empty at any given time. For example, a table on a 200 AMP system that defines 100,000 combined partitions with 100 rows per unaligned (127.5 KB) data block and 100 data blocks per each combined partition per AMP has 200 billion rows. This is a relatively small number of combined partitions when you consider that the maximum for a table or join index is 9,223,372,036,854,775,807 combined partitions.

If each row were 100 bytes in length, the primary data alone consumes 20 petabytes of disk. That is 20×10^{15} bytes. It is highly unlikely that every combined partition would be populated. When you consider a multidimensional use of multilevel partitioning, you can easily deduce that not all combinations of dimension values actually occur.

In general, a populated combined partition should have either many data blocks per AMP or no data blocks. For the example proposed in the first paragraph, if there is actually only 200 gigabytes of data and

each populated combined partition had 100 data blocks per AMP, about 99% of the combined partitions are empty.

Byte Alignment

Rows on packed64 format systems are always aligned on even-byte boundaries. In other words, rows are never stored with an odd number of bytes. As a result, if a row has an odd byte length, the system adds a filler byte to the end of the row to make its length even. Filler bytes are included in the CurrentPerm total for each table column in DBC.DatabaseSpace.

During a SysInit operation, Vantage reads several flags to determine whether a system should be configured to format its data in a packed64 format or in an aligned row format. Consult your Teradata support representative if you want to change the current setting for your system. Depending on the settings of the row format flags, a freshly initialized system formats its rows in packed64 format by default, while an upgraded system continues to format its rows in aligned row format as the default.

See the table in [Row Structure for Aligned Row Format Systems](#).

A data block contains rows from one table only, and any row from that table is completely contained within a given data block. In the packed64 row format, a row starts on a 2-byte boundary relative to the start of a data block. This ensures alignment of columns only for those tables where the maximum alignment requirement for any column is 2.

In the aligned row format, the maximum alignment requirement for a column in a table can be 1, 2, 4, or 8 bytes. The size of each row must be a multiple of the maximum alignment requirement to ensure that each row starts on a valid boundary within a data block. For the sake of uniformity, all row sizes are multiples of 8, starting on an 8-byte boundary relative to the start of a data block. This requires additional disk space for aligning data and the multiples of 8 filler applies to both small-block-sized systems and large-block-sized systems.

Write Ahead Logging (WAL) introduces a data block header size of 72 bytes. This increase causes 5.5% of existing data blocks to increase in size by one sector (512 bytes).

The 28 byte increase in size of the data block header plus the additional pad characters in the row data that are required to ensure 8-byte boundary alignment decreases the number of rows that can be stored in a data block with respect to the same data stored on pre-WAL systems.

The high-level structure of a file system data block is as follows.

Datablock Header	Compression Header	Row Data	Reference Array	Datablock Trailer
------------------	--------------------	----------	-----------------	-------------------

The size of the data block trailer is 2 bytes for small block-sized systems and 4 bytes for large block-sized systems. The size of the compression header is the same for small-block-sized systems and large-block-sized systems. The size of each element in the reference array is 2 bytes for all systems.

You should take this information into account when you undertake the various table sizing operations performed during capacity planning. This information can also be important if you need to determine the exact amount of compression you can achieve by compressing multiple column values.

Row Length Characteristics

The maximum length for a database row on both packed64 and aligned row format systems is 1 MB.

Row Components

The following table describes the individual components of a base table row that does not have column partitioning for both packed64 systems and aligned row systems:

Component	Number of Bytes	Description
Reference array pointer	2	The reference array pointer is not part of the physical row. It is a 2-byte entry maintained at the bottom of each data block that points to the first byte of each row in the block.
Row Header		
Row length, small-row format	2	Specifies the exact length of the row in bytes.
Row length, large row format	4	Specifies the exact length of the row in bytes.
Nonpartitioned Primary Index RowID	8	Contains two physical fields and one virtual field: <ul style="list-style-type: none"> • Partition number: Implicitly assumed to have a value of 0 (as indicated by 2 bits in the flag byte). Consumes 0 bytes from the row. • Row hash value • Uniqueness value
Row hash	4	Contains the hashing algorithm-generated row hash value for the row. The value is 4 bytes wide irrespective of the number of hash buckets the system has.
Uniqueness value	4	Contains the system-generated uniqueness value for the row.
Internal partition number	0	This field is logical, not physical. It is not stored as a physical value because its value is known to the system always to be 0.
Partitioned RowID	10 or 16	Contains these physical fields: <ul style="list-style-type: none"> • Partition number: Contains a value between 1 and 65,535 or big number that defines the internal partition number for the row. Consumes 2 or 8 bytes from the row. • Row hash value.

Component	Number of Bytes	Description
		<ul style="list-style-type: none"> Uniqueness value. <p>Note that the partition number field is not contiguous with the row hash and uniqueness value fields; instead, it immediately follows the first byte of the presence bits array.</p>
Row hash	4	Contains the hashing algorithm-generated row hash value for the row. The value is 4 bytes wide irrespective of the number of hash buckets the system has.
Uniqueness value	4	Contains the system-generated uniqueness value for the row.
Internal partition number	2 or 8	<p>Contains the partition number for a row in a PPI table.</p> <ul style="list-style-type: none"> If the maximum partition is $\leq 65,535$, partition number is 2 bytes. If the maximum partition is $> 65,535$, partition number is 8 bytes. If the row does not belong to a PPI table, then this field is logical only, and its value is 0.
Flag byte	1	<p>Defines whether the 4-byte Partition number field is used or not.</p> <ul style="list-style-type: none"> If the flag is set, the row is from a PPI table and the row header uses the 4-byte partition number field that follows the first presence octet to store its partition number. If the flag is not set, the row is not from a PPI table and the system assumes that the partition number for the rowID is 0.
Presence Bits Array		
First byte of the presence bit array See Presence Bits for more information.	<p>1</p> <p>If the table has a PPI, there is an additional 2 or 8 byte overhead in the presence bits array.</p>	Defines column nullability and compressibility.
Additional bytes of presence bit arrays	8 bytes per presence bit array.	<p>Provides additional space to define nullability and compressibility for columns if the first presence bit array is too small to contain all the information for the table.</p> <p>For more information, see Presence Bits.</p>
VARCHAR Offset Array		
Offset array pad bytes The array pad bytes are only used for aligned row format systems.	Varies	<p>Aligns the offset array at a 2-byte boundary.</p> <p>This component exists for rows on aligned row systems only.</p>

Component	Number of Bytes	Description
Column offsets	Varies <ul style="list-style-type: none"> For small-row format: 2 bytes per variable length column. For large-row format: 4 bytes per variable length column. 	Only present when variable length columns are defined for a table. Indicates the intrarow location of the column it references.
Load-Isolation Information		
RowLoadID	8	A value that records the committed property of a row in a load-isolated table. This component exists for load-isolated tables only.
Fixed Length Columns		
Fixed length column pad bytes ^b	Varies	Aligns the fixed length columns on an 8-byte boundary. This component exists for rows on aligned row systems only.
Fixed length columns	Varies	Contains all non-compressible fixed length columns.
Compress Length Columns		
Compressible column pad bytes ^b	Varies	Aligns the value compressible columns on an 8-byte boundary. This component exists for rows on aligned row systems only.
Compressible columns	Varies	Contains compressible columns with data belonging to one of four categories: <ul style="list-style-type: none"> Not compressed using multivalue compression. Not compressed using algorithmic compression. Compressed using algorithmic compression. Compressible data that is not null.
Variable-Length (VARCHAR and VARBYTE) Columns		
Variable length column pad bytes ^b	Varies	Aligns the variable-length columns on an 8-byte boundary. This component exists for rows on aligned row format systems only.
VARCHAR and VARBYTE columns	Varies	Only present when variable-length columns are defined for a table. Contains all variable length character columns. The column offsets field points to the starting location for each variable length column in the row.

Component	Number of Bytes	Description
Row Trailer		
Trailing pad bytes	Varies	Aligns the row on an even-byte boundary for packed64 format systems or on an 8-byte boundary for aligned row systems.

Hash and Join Index Row Structures

The row structure for non-row compressed hash and join indexes is the same as that for a standard data table (with the exception that you cannot compress individual column values for hash indexes (which means that the presence bit array of a hash index indicates only the nullability of a column, not its compressibility), while the row structure for row compressed hash and join indexes is more like that of a secondary index (see [Sizing a Unique Secondary Index Subtable](#) and [Sizing a Nonunique Secondary Index Subtable](#)).

Other than that exception, the only difference is the specific columns included in the base table data columns of the row. For all hash indexes, Vantage automatically adds the primary index columns of the base table (if not explicitly specified in the hash index definition) and its row uniqueness value. The system does not automatically add this information to join indexes.

Hash indexes are typically row compressed by default (see [Compression of Hash Index Rows](#) for details), while you must explicitly specify row compression in a CREATE JOIN INDEX request to compress rows in the join index it creates.

Note that you cannot row compress either of the following hash and join index types:

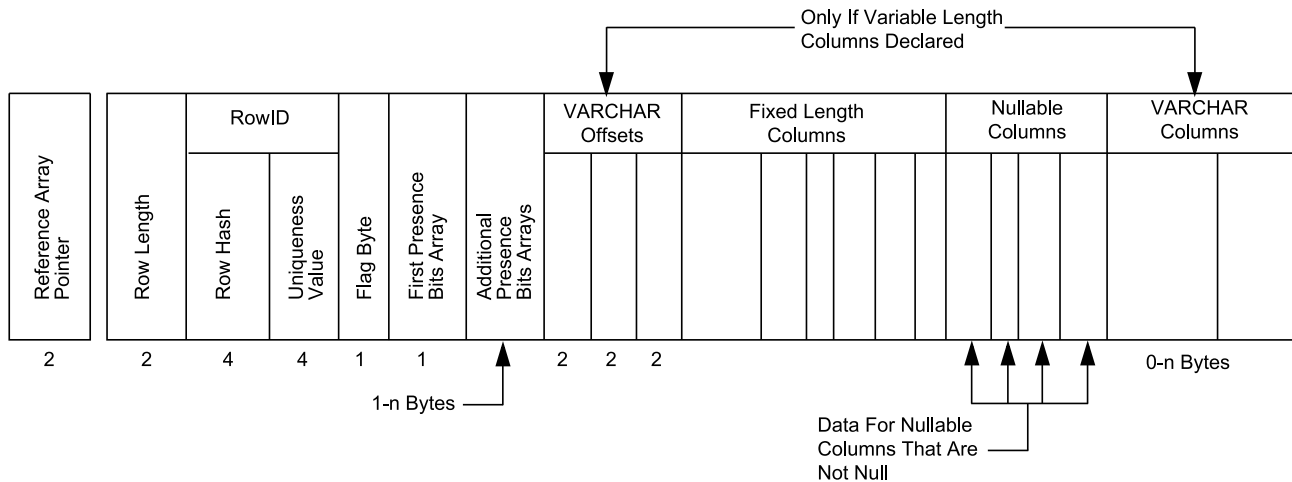
- A hash index defined with a PPI.
You cannot define any hash index with a partitioned primary index.
- A join index defined with a PPI or with column partitioning.

Hash and Join Index Row Structure for Packed64 Format Systems

Hash and join index rows on packed64 format systems do not have to align on 8-byte boundaries. Because of this, their row structure is simpler than that of equivalent base table rows on aligned row format systems (see [Row Structure for Aligned Row Format Systems](#)).

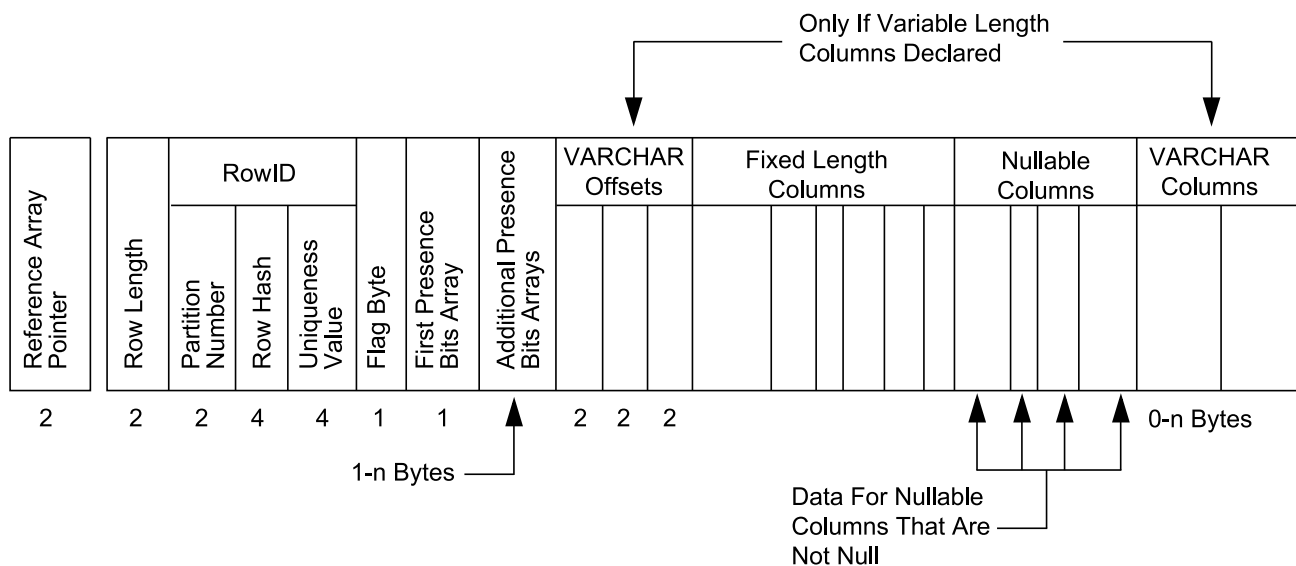
Packed64 Row Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index

The following graphic illustrates the basic structure of a non-row compressed hash or join index row from an index defined with an nonpartitioned primary index.



Packed64 Row Structure for an Uncompressed Load-Isolated Join Index With a Partitioned Primary Index

The following graphic illustrates the basic structure of a non-row compressed join index row (hash indexes cannot have a partitioned primary index) from an index defined with a partitioned primary index.



The difference between this and the format of a non-partitioned primary index row is the presence of a 2-byte partition number field at the beginning of the RowID. It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index indexes, the partition number is assumed by default to be 0, and is not stored.

For more information on join indexes and load isolation, see [Rules for Using Join Indexes on Load-Isolated Tables](#).

Packed64 Row Structure for a Row Compressed Hash or Join Index With an Nonpartitioned Primary Index

The region labeled as Index Value in the graphic of the row format is the *column_1* value set for the row. It is an abbreviation for the various data type orderings and offsets shown in detail in the row format diagrams for non-row compressed hash and join index rows.

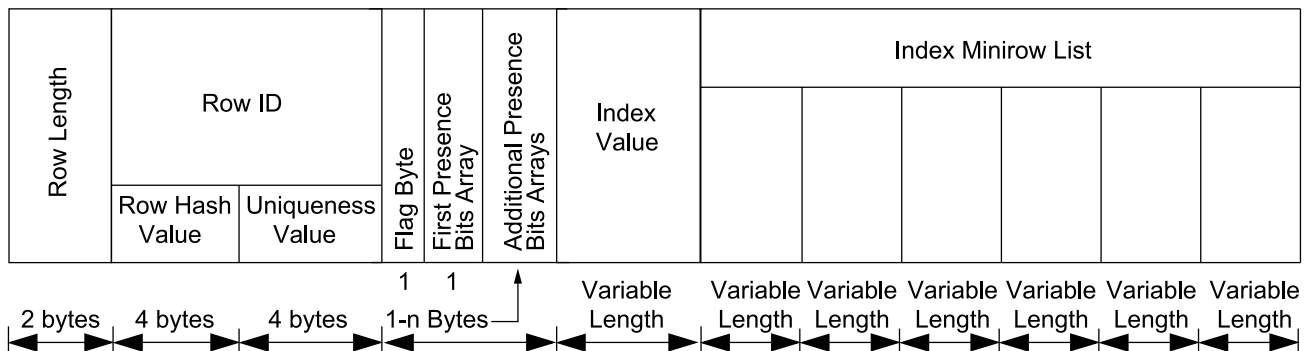
For example, consider the following example CREATE JOIN INDEX text:

```
CREATE JOIN INDEX test AS
  SELECT (a,b), (x,y)
  FROM table1, table2;
```

The system packs columns a and b in the last subfield of Field1 of a row compressed join index row the same way that index keys (an *index key* is the set of values stored as “the index” in a secondary index) are packed in the Field1 of a secondary index row (columns a and b are stored in the area labeled as Index Value in the row format graphic).

The system also stores the uncompressed index values (each instance of those (x,y) values that has the same (a,b) values as a minirow in Field2) in the area labeled as Index Minirow List in the row format graphic.

Each minirow consists of a BYTE length field, an optional presence bits array field, and a *column_2_name* value set as described above in *Packed64 Row Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index*, or *Packed64 Row Structure for an Uncompressed Load-Isolated Join Index With a Partitioned Primary Index*.



Packed64 Row Structure for a Hash or Compressed Join Index With a Partitioned Primary Index

Vantage does not support PPIs for hash indexes, compressed join indexes, or column-partitioned join indexes.

Hash and Join Index Row Structure for Aligned Row Format Systems

The hash and join index row structure diagrams for aligned row format systems differ from those of packed64 format systems by having as many as five additional pad byte fields (depending on the data types of the

columns defined for Index Value) to ensure row alignment on an 8-byte boundary. The following table lists each of these pad fields and explains their purpose.

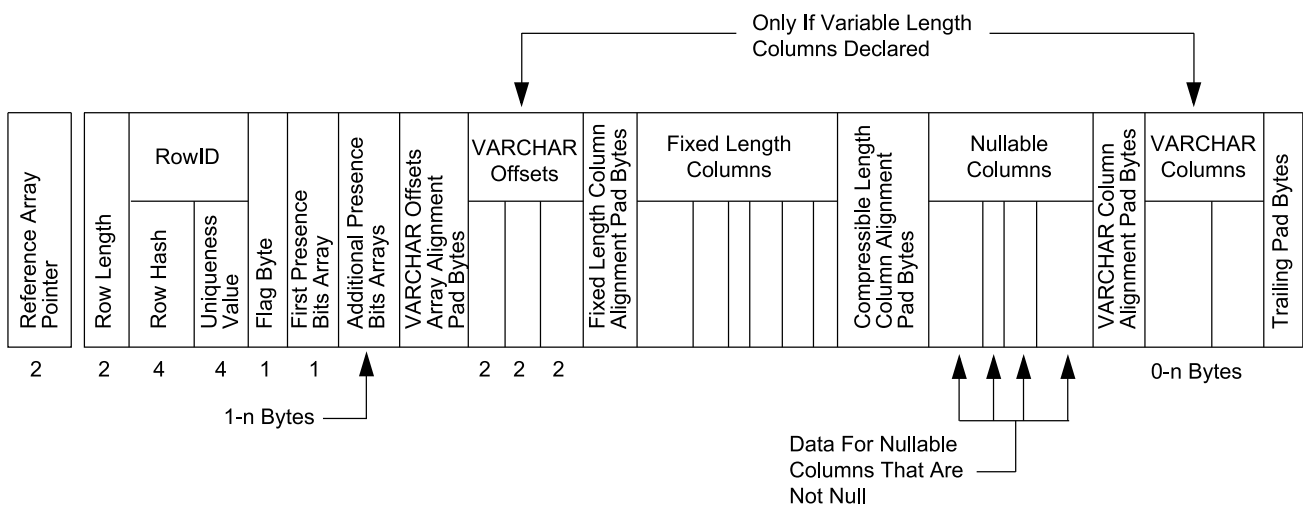
Pad Byte Field Name	Purpose
VARCHAR Offsets Array Alignment Pad Bytes	Aligns VARCHAR offsets array at a 2-byte boundary.
Fixed Length Column Alignment Pad Bytes	Aligns fixed length columns.
Nullable Length Column Alignment Pad Bytes	Aligns nullable length columns.
VARCHAR Column Alignment Pad Bytes	Aligns variable length columns.
Trailing Pad Bytes	Aligns entire row on an 8-byte boundary.

The index columns in a row compressed hash or join index row are stored in packed64 format, adjusted for aligned row format systems by a field of alignment bytes trailing field1 and another field of alignment bytes trailing field2 (if necessary to make the entire row align on a modulo(8) boundary). When you row compress a hash or join index, the *column_2_name* value set for each row in the index is stored as a minirow. Each minirow consists of a byte length field, an optional presence bits array field, and the *column_2_name* value set for the minirow.

Vantage does not support PPIs for hash indexes, row-compressed join indexes, or column-partitioned join indexes.

Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index

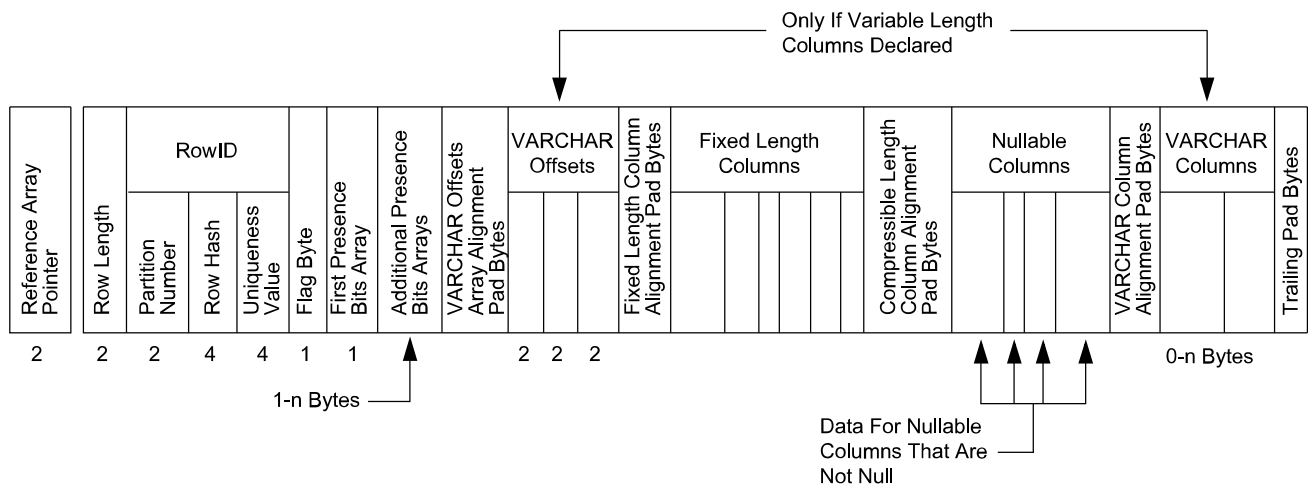
The following graphic illustrates the basic structure of a database row from a non-row compressed hash or join index with an nonpartitioned, or standard, primary index:



Aligned Row Format Structure for an Uncompressed Load-Isolated Join Index With a Row-Partitioned Primary Index and 65,535 or Fewer Combined Partitions

Vantage does not support PPIs for hash indexes.

The following graphic illustrates the basic structure of a database row from a non-row compressed load-isolated join index (hash indexes cannot have partitioned primary indexes) with a row-partitioned primary index (and no column partitioning):



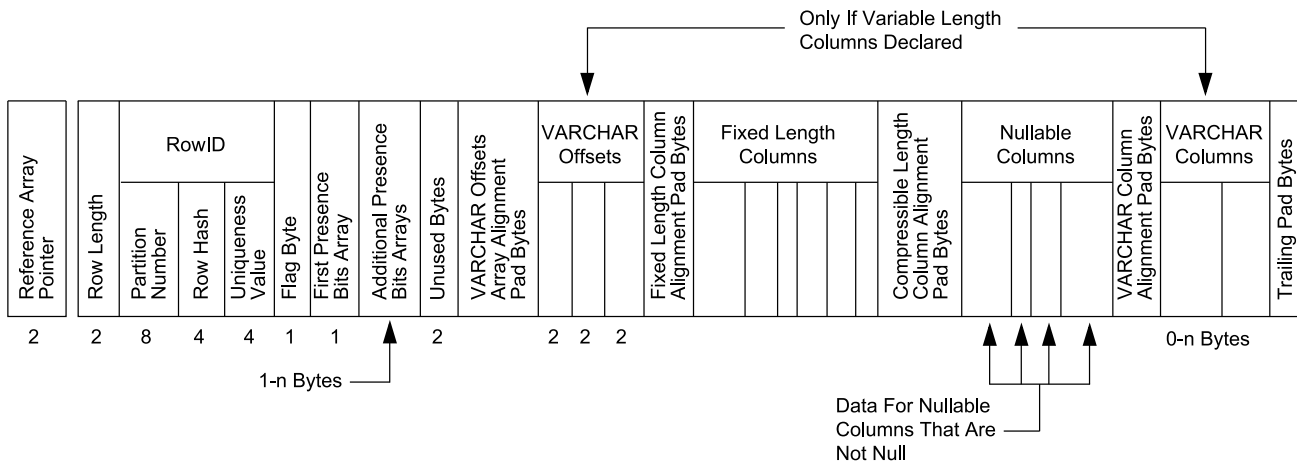
The difference between this and the format of a non-row compressed nonpartitioned primary index row is the presence of a 2-byte partition number field at the beginning of the RowID. It is this field that generates the need for a BYTE(10) data type specification for a RowID. For nonpartitioned primary index tables, the partition number is assumed to be 0, so the rowID of a nonpartitioned primary index table is also logically BYTE(10) (see [ROWID Columns](#)).

For more information on join indexes and load isolation, see [Rules for Using Join Indexes on Load-Isolated Tables](#).

Aligned Row Format Structure for an Uncompressed Join Index With a Partitioned Primary Index and More Than 65,535 Combined Partitions

Vantage does not support PPIs for hash indexes.

The following graphic illustrates the basic structure of a database row from a non-row compressed join index (hash indexes cannot have partitioned primary indexes) with a row-partitioned primary index (without column partitioning):



The difference between this and the format of a non-row compressed non-partitioned primary index row is the presence of an 8-byte partition number field at the beginning of the RowID. It is this field that generates the need for a BYTE(16) data type specification for a RowID.

Note:

The figure shows a row in a join index table without load isolation. If the join index is load isolated, an 8-byte RowLoadID is added after the VARCHAR offsets.

Aligned Row Format Join Index Row Layout for a Compressed Hash or Join Index With an Nonpartitioned Primary Index

Consider the following example CREATE JOIN INDEX text:

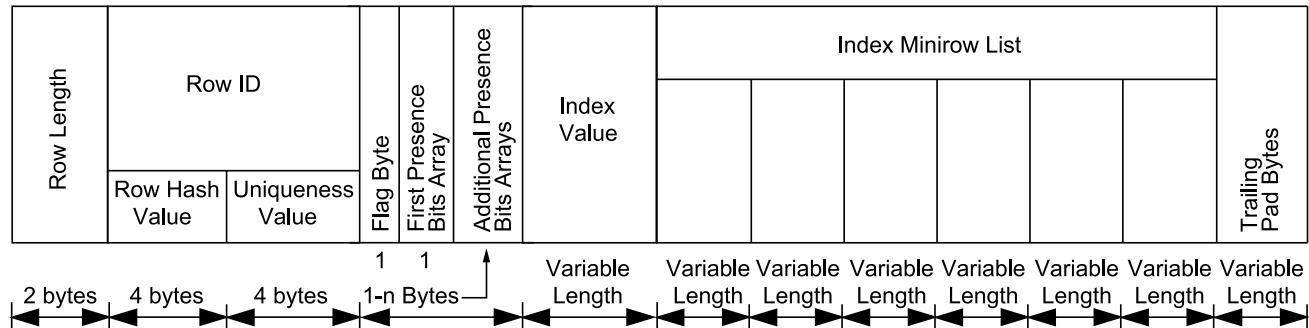
```
CREATE JOIN INDEX test AS
  SELECT (a,b), (x,y)
  FROM table1, table2;
```

Vantage packs columns a and b, labeled as the *column_1_name* list in the CREATE JOIN INDEX syntax diagram, in the last subfield of Field1 of a row compressed join index row the same way that index keys (an *index key* is the set of values stored as “the index” in a secondary index) are packed in the Field1 of a secondary index row (columns a and b are stored in the area labeled as Index Value in the row format graphic).

The system also stores the non-row compressed index values (each instance of those (x,y) values that has the same (a,b) values as a minirow in Field2) in the area labeled as Index Minirow List in the row format graphic. These columns are labeled as the *column_2_name* list in the CREATE JOIN INDEX syntax diagram.

The region labeled as Index Value in the following row format graphic is the *column_1_name* value set for the row and should be interpreted as containing the requisite pad bytes depicted in [Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index](#). It is an

abbreviation for the various data type orderings and offsets shown in detail in the row format diagrams for non-row compressed hash and join index rows. The number of trailing pad bytes required by a given row is the number of bytes it takes to make the row end on a modulo(8) boundary.



When you row compress a hash or join index, the `column_2_name` value set for each row in the index is stored as a minirow. Each minirow consists of a BYTE length field, an optional presence bits array field, and a `column_2_name` value set (see [Aligned Row Format Structure for an Uncompressed Hash or Join Index With an Nonpartitioned Primary Index](#) or [Packed64 Row Structure for a Row Compressed Hash or Join Index With an Nonpartitioned Primary Index](#)).

Aligned Row Format Join Index Row Layout for Compressed Hash and Join Indexes With a Partitioned Primary Index

Vantage does not support PPIs for hash indexes, row-compressed join indexes, or column-partitioned join indexes.

Presence Bits

Note:

Much of the following discussion does not apply to column-partitioned tables and join indexes except as it applies to column partitions with ROW format and to multi-column partition values in containers.

Presence bits indicate the status of each column with respect to its nullability, multivalue compressibility, and autocompressibility. Compression presence bits are added to the row header of each row to specify how multivalue compression is used for that row.

Each row has at least one octet of presence bits and can have more, depending on the degree of the table and the cumulative number of values compressed. The difference between bytes and octets is conceptual. The 8 bits of a byte define an atomic unit, while each individual bit of an octet is an atomic flag in a bit array. For the purpose of storage capacity analysis, both are treated as bytes. All rows for a given table have the same presence bits defined because nullability and multivalue compressibility are table attributes.

The first bit of the first presence bits octet is always set to 1, so the first octet defines the nullability and multivalue compressibility for no more than seven columns. When necessary, additional presence bit octets are added to the row header. Although only 7 of the bits in the first octet are available for use as presence

bits, all 8 bits of successive octets are available. Eight bits are used because 255 compressed values plus the indication that an uncompressed value is present require 2^8 , or 256 bit combinations, to be represented.

For a column-partitioned table or join index there is one set of presence bits for each column partition value in the container. If a presence bit is 1, a value is present. If a presence bit is 0, no value is present.

A compression-enabled column for a table has a maximum of 256 presence bits set, depending on how many values are compressed using multivalue compression.

The meaning of the number of presence bits set per column for compression of a single value is provided in the following table.

This type of presence bit field...	Has this many presence bits ...	For this type of column ...
Nullability	0	Non-nullable.
	1	Nullable.
Compressibility	0	Not compressed or compressed on nulls only.
	1	Compressed on a value.

The total number of presence bits for a given row is the sum of the nullability presence bits and the compressibility presence bits.

The following table expresses the same information in a slightly different way.

Compressible		Nullable	
Bit Value	Meaning	Bit Value	Meaning
0	The column is multivalue compressed.	0	The column is null.
1	A non-compressed column value is present.	1	The column is not null.
An algorithmically compressed column adds an extra bit to indicate whether the column is algorithmically compressed or not, as follows:			
Compressible		Nullable	
Bit Value		Meaning	
0		The column is not algorithmically compressed.	
1		The column is algorithmically compressed.	
1 to 8 bits for each multivalue compressible column (8 bits because 255 compressed values plus null require 28, or 256 bit combinations, to be represented).		1 bit for each nullable column.	

The following table provides a comprehensive mapping of the presence bits and their various combinations for the multivalue compression case:

WHEN the presence bits for a column have these values ...		THEN the column ...	AND ...
Compress	Null		
no bit	no bit	is not compressible	is not nullable.
0	no bit	is compressed	is not nullable.
1	no bit	contains uncompressed column values	is not nullable.
no bit	0	is not compressible	is null.
no bit	1	is not compressible	is not null.
0	0	is compressed	is null.
1	1	is not compressed	is not null.
1	0	is not compressed	is null.
0	1	is not compressed.	is null.

Mappings of COMPRESS bit values for multivalued compression generalize from this specific case as illustrated by the following table.

IF the presence bit is ...	THEN the data is ...
1	not compressed. The corresponding compress bits are all 0.
0	compressed. <ul style="list-style-type: none"> If the corresponding compress bits are all 0, then the compress value is null. If the corresponding compress bits are not all 0, then the compress value is an index to the compress multivalue array in the table header.

The following table presents a set of examples that clarifies the correspondence between presence bits and data attribute specifications.

FOR this column definition ...	The presence bits are ...	For these characters ...
col_1 CHAR(1) NOT NULL	none	A
col_1 CHAR(1) NOT NULL COMPRESS ('A')	0	A
	1	B
col_1 CHAR(1) COMPRESS	0	null
	1	A

FOR this column definition ...	The presence bits are ...	For these characters ...
col_1 CHAR(1) COMPRESS ('A')	00	null
	01	A
	10	B
	10	C
col_1 CHAR(1) COMPRESS ('A', 'B', 'C', 'D')	0000	null
	0001	A
	0010	B
	0011	C
	0100	D
	1000	E
col_1 CHAR(1) NOT NULL COMPRESS ('A', 'B', 'C', 'D')	001	A
	010	B
	011	C
	100	D
	000	E

Number of Presence Bits Required to Represent Compressed Values

Because compression is signaled by populating octets of presence bit fields in the row header, there is a tradeoff between the capacity required to add presence octets and the savings realized from multivalued compression at the base table row level: the size of the bit field required to express multivalued compression increases nonlinearly as a function of the number of values to compress. With respect to the savings gained by multivalued compression, multivalued compression is almost always worth doing.

The following table shows the relationship between the number of values specified for compression in each column and the number of bits required to indicate that number of values. Null is always one of the values compressed when multivalued compression is specified for a nullable column, even if null is not explicitly listed in the COMPRESS clause.

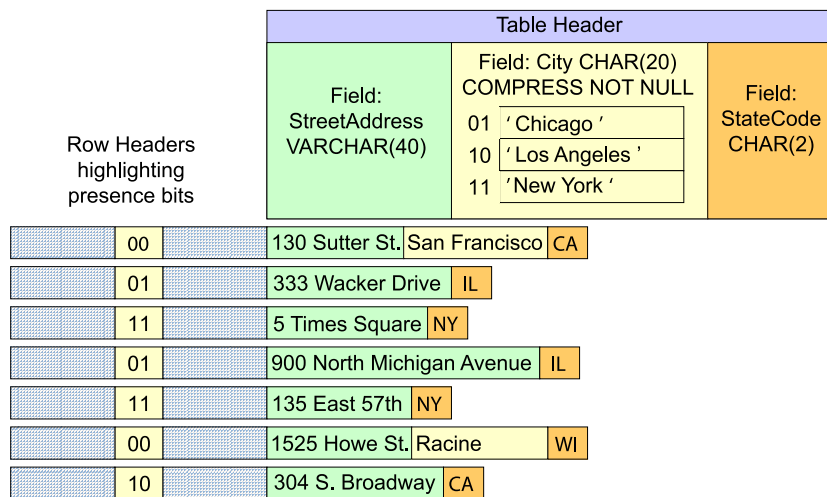
Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
Yes	null	1 Shared with the presence bit.
No	1	1

Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
No	2 - 3	2
No	4 - 7	3
No	8 - 15	4
No	16 - 31	5
No	32 - 63	6
No	64 - 127	7
No	128 - 255	8
Yes	1 (including null)	1
Yes	2 - 3 (including null)	2
Yes	4 - 7 (including null)	3
Yes	8 - 15 (including null)	4
Yes	16 - 31 (including null)	5
Yes	32 - 63 (including null)	6
Yes	64 - 127 (including null)	7
Yes	128 - 255 (including null)	8

The compress bits value represents an index into the compress multivalue array for that column in the table header. The following graphic shows that the presence bit pattern 10, for example, references the compressed character string “Los Angeles” in the table header. Because only 3 values are compressed for this column, 2 compression bits are required to uniquely identify them.

The table definition DDL for this illustration is as follows:

```
CREATE TABLE MVCompress (
  StreetAddress VARCHAR(40),
  City          CHARACTER(20) COMPRESS ('New York',
    'Los Angeles', 'Chicago') NOT NULL,
  StateCode     CHARACTER(2));
```



As you can see, all the possible combinations of bit patterns are used to identify and locate the values. For this particular example, the bit patterns and their corresponding values are those indicated in the following table:

FOR this pattern of presence bits ...	The value for City is ...	AND stored in ...
00	not compressed.	the row.
01	'Chicago' and is compressed	the table header.
10	'Los Angeles' and is compressed	the table header.
11	'New York' and is compressed	the table header.

From the perspective of the row header, the best policy is to compress the highest number of values in each octet bit array, because doing so does not add any more space in the row header. In other words, the best practice is to optimize multivalued compression for 1, 3, 7, 15, 31, 63, 127, or 255 values. You should evaluate all these possibilities to determine which yields the best compression for a particular column.

You must also take into account the number of bytes compression transfers to the table header, because those bytes count against the maximum row length of 1 MB.

Note:

Systems using small cylinders have a maximum row length of 64 KB.

The compressed values are stored adjacent to one another in ascending value order and descending order of field size following the end of Field5 in the table header.

Table Headers

Each database table has an associated subtable that contains the table header (table metadata). One copy of the table header is stored on each AMP that stores rows of the table. The map associated with the table at the time the table was created or altered specifies which AMPs store the table rows.

Vantage uses table headers internally to maintain various information about each table. Table headers are described here primarily because they are used to store value-compressed column values (see [Compression Types Supported by Vantage](#)), and it is difficult to explain how compression works without having at least a superficial understanding of this file structure.

The size limit for a table header is 1 MB, so it is not possible to create a table defined with all possible features (maximum numbers of columns, indexes, and compressed values) without exceeding the defined byte limit. The maxima stated in [Teradata System Limits](#) are for the individual table parameters only.

Table Header Components

The following topics describe the individual fields of a Vantage table header as it is stored on disk. The maximum length for each field is the value for a thin table header.

Row Header

See [Base Table Row Format](#) for a description of row headers.

Field 1

Fixed length. Includes:

- Row 1 header.
- Offset array for the variable length columns in the table header, Used to locate those columns.
- Table header row format version.
- Internal ID of the database to which this table belongs.
- Internal ID of the database to which space for this table is charged.
- Table creation timestamp.
- Last table update timestamp.
- Last table archive timestamp.
- Primary index flag.
- Table structure version. Updated each time the table description is modified
- Table structure version for host utilities. Incremented each time a structural change is made that makes a database dump obsolete.
- Number of backup tables associated with this table.
- Internal ID of the permanent journal table.
- Table kind. Describes whether the table is permanent, temporary, volatile, or a join index. For temporary tables, volatile tables, and join indexes, also describes preserve-on-commit and transient journaling characteristics.
- Journal type:

- After
- Audit
- Both
- Before
- None
- Protection type:
 - Fallback
 - Log
 - None
- User journal flag. Describes whether the table is a user-defined journal or not.
- Hash flag. Describes whether table is hashed or not.
- Dropped flag. Set TRUE from the time the AMP receives a Drop Table step until the table is dropped in the End Transaction step. Otherwise set FALSE.
- DDL change flag. Prevents attempts to update the table after its DDL has changed. Set TRUE when table DDL is modified.

Host utility table dump sets the flag FALSE.

- Byte count of the number of USIs defined on the table.
- Host character set at the time the table was created.
- Number of parent tables referenced by this table.
- Number of child tables referencing this table.
- Merge block ratio for the table.
- Merge block ratio validity. Indicates whether the specified merge block ratio for the table is valid or unspecified.
- Data block size for the table in bytes.
- Data block size validity. Indicates whether the specified data block size for the table is valid or unspecified.
- Percent free space for the table.
- Percent free space validity. Indicates whether the specified percent free space for the table is valid or unspecified.
- Disk I/O integrity checksum. Used to verify the integrity of user tables, hash indexes, join indexes, and secondary indexes.
- Message class of primary step.
- Message kind of primary step.
- Message class of secondary step.
- Message kind of secondary step.
- Host ID.
- Session number.
- Request number.

- Transaction number.
- Table ID of base temporary table. Used only for materialized temporary tables.
- Internal ID for primary key index.
- Restart flag. Tracks restart and non-restart cases of restore jobs during build phase.
- Row format, which specifies whether current environment is packed64 format or aligned row format.
- Dummy space.
- List of index descriptors for the table in Index ID order, one per primary and secondary index defined on the table.
- Row 1 length. Duplicate specification of the length of row 1.
- Map number.
- Map slot.
- 1 byte to indicate whether map is contiguous or sparse.

Field 2

Variable length.

Maximum length: 47,784 bytes.

Contains the primary index descriptor and all secondary index descriptors for the table.

Field 3

Not used.

Field 4

Variable length.

Maximum length: 423 bytes.

Contains MultiLoad, FastLoad, and table rebuild information.

Field 4 is always present for permanent journal tables, but is context-dependent for non-journal tables.

Field 5

Variable length.

Maximum length: 56,350 bytes.

Contains the following table column descriptors:

- Internal ID of the first column in the table.
- Number of varying length columns in the table.
- Number of presence bits in each row. The upper limit for the number of presence and compress bits per row is 89,991.
- Flag to indicate whether the table header is thin or fat. Used to determine whether the compressed value offsets list contains actual addresses or must be left-shifted to extract actual addresses.
- Compression flag. Specifies whether the table has compressible columns or not.

- Offset in the row to the presence bit array. Presence octet locations are determined by dividing the presence bit position by 8.
- Offset in the row to the first byte past the presence bit array.
- Number of columns in the table.
- Index into the field descriptor array to the first compressible column. If no columns are compressed, then the value points to the first varying length column in the row.
- Offset from the beginning of the row to the first optional (varying or compressible) data.
- Index of the field descriptor for the first physical column in the row.
- Field 5 type:
 - Table descriptor with row hash and unique rowID.
 - Index descriptor with row hash and unique rowID in the index.
 - Table descriptor with a partitioned rowID.
 - Index descriptor with a partitioned rowID in the index.
- Row format. Specifies whether current environment is packed64 format or aligned row format.
- Duplicate rows flag:
 - Dictionary and non-ANSI/ISO tables.
 - ANSI/ISO tables without unique indexes.
 - ANSI/ISO tables with unique indexes.
- Offset to system code to build rows and to calculate PPI internal partition numbers.
- Field descriptors array.
- Compressed values and UDT contexts. Compressed values and a UDT context are stored just beyond the Field 5 descriptors and partition-related system code. The stored values are sorted first in ascending order of their binary values, then in descending order of their field size, and aligned on 2-byte boundaries. The stored UDT context is its autogenerated UDF constructor context. The area contains a 76-byte UDT context for every column typed with a UDT. This places a practical upper limit of approximately 1,600 UDT columns per table.
- System code for building rows, including code to calculate partition numbers for the rows of row-partitioned tables. The row-partition-related system code is stored after the Field 5 descriptors, and just before the compressed values, if any have been defined.
- UDT name stored as a variable length string of up to 128 Unicode characters.

Field 6

Variable length. Usually null.

Maximum length: 118 bytes.

Contains restartable sort and ALTER TABLE information.

Field 7

Variable length.

Maximum length: 94,592 bytes.

Contains:

- Up to 128 reference index descriptors: 64 from a parent table to child tables and 64 from child tables to a parent table. See [Sizing a Reference Index Subtable](#) for a description of reference indexes.
- List of the names of unresolved child tables from referential integrity constraint specifications as a variable length string of up to 128 Unicode characters.

Field 8

Variable length.

Maximum length: 520 bytes.

Contains BLOB, CLOB, and XML descriptors.

Field 9

Variable length.

Maximum length: 518 bytes.

Contains:

- Length of, and offset to, the database name.
- Length of, and offset to, the table name.
- Database name (up to 128 Unicode characters).
- Table name (up to 128 Unicode characters).

The database and table names are in the format *databaseName.tableName*.

Field 10

Variable length.

No maximum length.

Description:

- Indicates if any row-id ranges, or regions, of any data subtables have been marked down or invalid.
- Each data subtable could have up to 6 invalid regions before the entire data subtable is marked invalid.

Field 11

Fixed length.

Maximum length: 82 bytes.

Description:

- Indicates if the protected table is a FastLoad, MultiLoad, or partitioned restore error table.
- Displays an array of field IDs of security constraint columns.
- Displays an array of constraint IDs associated with the constraint columns.

- Contains timestamps when the constraints were last altered.

Field 12

Additional information about the map the table uses, including:

- Number of AMPs in the map.
- Starting AMP number.
- For sparse maps, colocation name.

Sizing Structured UDT Columns

For a distinct UDT, there is always a 1:1 correspondence between the type and its underlying predefined data type. As a result, to determine the size of a distinct UDT column, just use the size of the underlying predefined data type. Because of alignment differences for packed64 and aligned row systems, be sure to consult the size differences for the various data types on the two system types before making your row size estimates (see [Data Type Size Differences For Packed64 and Aligned Row Format Architectures](#)).

Unlike predefined data types, the size of each structured UDT you create is different, and you must calculate that size to account for the amount of storage a column with that type requires.

Note:

You cannot use either multivalue compression or algorithmic compression for structured UDT column data.

This topic first describes how a structured UDT is stored in a row, then describes how to calculate the number of bytes required to store a given structured UDT in persistent form.

The dynamic UDT data type is a structured UDT with a fixed data type name of `VARIANT_TYPE`, so you can calculate the sizes of `VARIANT_TYPE` instances in the same way you would any other structured UDT.

Storage Structure of a Structured UDT Data Type

The following graphic illustrates the basic morphology of a structured UDT as it is stored in the row of a table in the database:

Type ID
Presence Byte Array
Attribute 1 Value
...
Attribute n Value

1094A053

where:

UDT element ...	Specifies ...
Type ID	the TVMID type identifier for the structured UDT. Its data type is INTEGER (4 bytes).
Presence Bits Array	the octet array of presence bits for the UDT at level <i>m</i> of the structured UDT. This is a variably-sized bit array, rounded to the higher modulo(8) boundary, whose size depends on the number of attributes stored for a particular structured UDT. Its size increases in octet (8-bit byte) increments.
Attribute <i>n</i> Value	attribute number <i>n</i> for the structured UDT. There is a variable number of attribute value fields. The exact number depends on two factors: <ul style="list-style-type: none"> • The number of attributes a particular structured type has. • Whether or not an attribute is null. The exact format of each attribute depends on the attribute type and is defined by the designer of the given structured type.

Storage Structure of a Nested Structured Data Type

A structured UDT can be built from attributes that are themselves structured UDTs. This topic explains the persistent storage format for a column having such a type.

The following graphic illustrates the basic morphology of a structured UDT that contains nested structured UDT attributes as it is stored in the row of a table in the database:

Type ₀ ID
Presence Bits Array ₀
Attribute 1 ₀ Value
...
Attribute n ₀ Value
Type ₁ ID
Presence Bits Array ₁
Attribute 1 ₁ Value
...
Attribute n ₁ Value
...
Type ₅₁₁ ID
Presence Bits Array ₅₁₁
Attribute 1 ₅₁₁ Value
...
Attribute n ₅₁₁ Value

where:

UDT element ...	Specifies ...
Type m ID	<p>the TVMID type identifier for the level m structured UDT.</p> <p>The value of m is 0 for the highest level attribute set in a structured UDT, ranging to 511 for the lowest possible level attribute set in a multilevel nested structured UDT.</p> <p>Structured UDT attributes can be nested through multiple levels. The lowest level is numbered 511 because there can be 512 attribute nesting levels in a structured UDT, ranging from 0 - 511.</p>
Presence Bits Array m	<p>the octet array of presence bits for the UDT at level m of the structured UDT.</p> <p>This is a variably-sized bit array, rounded to the higher modulo(8) boundary, whose size depends on the number of attributes stored for a particular structured UDT. Its size increases in octet (8-bit byte) increments.</p>
Attribute n_m Value	<p>attribute number n for the level m structured UDT component.</p> <p>The number of attribute value columns is variable. The exact number depends on two factors:</p> <ul style="list-style-type: none"> • The number of attributes a particular structured type has. • Whether or not an attribute is null. <p>The exact format of each attribute depends on the attribute type and is defined by the designer of the given structured type.</p>

About Sizing a Structured UDT

Because the size of a structured UDT value is not an arithmetic sum of the size of its individual attributes, it presents a special problem for capacity planning.

The storage footprint for a structured UDT is composed of the following components in the order given:

1. The TVMId Type Identifier for the UDT.
2. A variably sized attribute presence bit array with one bit per attribute, rounded up to the nearest 8-bit boundary.
3. A serialized list of the non-null attribute value sizes.

Example 1: Calculating the Storage Requirements of a Two-Level Structured UDT on a Packed64 Format System

What follows is a simple example of calculating the storage requirements of a two-level structured UDT on a packed64 format system:

Suppose you have the following nested structured type.

Level	Number of Attributes At The Level	Data Types of the Attributes
0	5	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER • Structured UDT
1	3	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER

Here is what is stored for a value having this structured type, assuming there are no null attributes. Nothing is stored to represent a null attribute other than a bit in the Presence Bits Array, which means that there is no difference in the storage of a compressed null and an uncompressed null. This means that with respect to compression, there is only one storage state for nulls, and that state is referred to as compressed. Because the representation of the states is identical, it is often said that nulls are compressed by default, but this is somewhat misleading.

- 6 bytes for the TVMID for level 0, stored in INTEGER format.
- 1 octet (8-bit byte) byte for the Presence Bits Array for level 0, which contains 5 presence bits for the 5 attributes at level 0 (all set to 1) and 3 unused presence bits (all set to 0).
- 4 bytes for INTEGER value 1 at level 0.
- 4 bytes for INTEGER value 2 at level 0.
- 4 bytes for INTEGER value 3 at level 0.
- 4 bytes for INTEGER value 4 at level 0.
- Size in bytes of the level 1 structured UDT, which is:

- 6 bytes for the TVMID for level 1, stored in INTEGER format.
- 1 byte for the Presence Bits Array for level 1, which contains 3 presence bits for the 3 attributes at level 1 (all set to 1) and 5 unused presence bits (all set to 0).
- 4 bytes for INTEGER value 1 at level 1.
- 4 bytes for INTEGER value 2 at level 1.
- 4 bytes for INTEGER value 3 at level 1.

Note that there is overhead of a TVMID value and a Presence Bits Array for each nesting level in a structured UDT.

For an aligned row format system, you just take the calculated size for a packed64 format system modulo(8) to align the column on an 8-byte boundary.

Example 2: Packed64 System

Now consider the following more concrete example, again for a packed64 system.

Suppose you create two structured types, one of which is an attribute of the other, as follows:

```
CREATE TYPE name_udt AS (
  first_name VARCHAR(20),
  last_name  VARCHAR(20));

CREATE TYPE address_udt AS (
  street  VARCHAR(20),
  city    VARCHAR(20),
  zipcode INTEGER,
  name    NameUdt);
```

For the sake of this example, assume the TypeID for name_udt is 33 and the TypeID for address_udt is 999.

Suppose you insert the following data into a column typed as address_udt:

```
INSERT INTO test_table
VALUES (NEW address_udt().street('Apple Tree Way')
      .city('Washington D.C.')
      .zipcode(10776)
      .name
      (NEW name_udt() .first_name('Abraham')
      .last_name('Lincoln')));
```

The nested structured type address_udt, which nests the UDT name_udt as one of its attributes, is stored as indicated by the following graphic.

999
00001111
14, 'Apple Tree Way'
15, 'Washington D.C.'
10776
33
00000011
7, 'Abraham'
7, 'Lincoln'

System-Derived and System-Generated Columns

About System-Derived Columns

As the name implies, system-derived columns are columns whose values are not created by users, but instead are derived dynamically by the system. Some system-derived columns contain values that Vantage creates and maintains for internal use, while others contain values that are critical for user applications.

The following system-derived column types are supported by Vantage:

- ROWID columns
- PARTITION and PARTITION#L *n* columns

ROWID, PARTITION, and PARTITION#L *n* columns are always system-defined and their values are always system-generated.

About System-Generated Columns

In some cases of system-generated columns the DBA specifies the name of the column and whether or not it is to be created for a table and in other cases, the existence, name, and contents of the column are all system-controlled.

The following system-generated column types are supported by Vantage:

- Identity columns

Identity columns are user-defined, but Vantage defines the values inserted into them (in the case of GENERATED ALWAYS AS IDENTITY columns, all column values are system-generated. In the case of GENERATED BY DEFAULT AS IDENTITY columns, inserted column values can be user-generated or system-generated.).

- Object Identifier columns (see [Object Identifier Columns](#))

Similarly, BLOB, CLOB, and XML columns are user-defined, but Vantage defines the OID values that point to them.

ROWID Columns

Every base table, join index, and hash index has a system-generated column named ROWID. The fields in this column contain the RowID value for their rows.

FOR a partitioned table or join index that has ...	The data type for ROWID values is ...
65,535 or fewer combined partitions	BYTE(10)
> 65,535 combined partitions	BYTE(16)

The system-derived column ROWID contains the internal row identifier associated with a row of a base table or join index.

With one exception, there is nothing different for a column-partitioned table or join index. The exception is that the column partition is always 1 for the internal partition number in the ROWID of a column-partitioned table or join index. If you only specify column partitioning when you create a column-partitioned table or join index and do not specify the ADD option, the table or index always uses 2-byte partitioning.

As a user, you can only specify the ROWID keyword in a CREATE JOIN INDEX request to enable non-covering join indexes to join with base table columns to optimize query processing (see [Partial Query Coverage](#), [Restrictions on Partial Covering by Join Indexes](#), and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144). You cannot specify ROWID in any other context at any time.

The rules for using the ROWID keyword in a CREATE JOIN INDEX request are as follows.

- You can optionally specify the ROWID for a base table in the select list of a nonpartitioned or PPI join index definition.

The select list for a column-partitioned join index must include the system-derived column ROWID of the base table, and it must be specified with an alias.

If you reference multiple tables in the join index definition, then you must fully qualify each ROWID specification.

- You can reference an alias for ROWID, or the keyword ROWID itself if no alias name has been specified for it, in the primary index definition or in a secondary index defined for the join index in its index clause.

This does not mean that you can reference a ROWID or its alias in the DDL you use to create a secondary index defined separately from CREATE TABLE using CREATE INDEX (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144) after the join index has been created.

- If you reference a ROWID alias in the select list of a join index definition, then you can also reference that correlation name in a CREATE INDEX request that creates a secondary index on the join index.

- Aliases are required to resolve any column name or ROWID ambiguities in the select list of a join index definition. An example is the situation where you specify ROWID for more than one base table in the index definition.
- Aliases are mandatory for a ROWID specification in a column-partitioned join index.

If you attempt to use the ROWID keyword in any other context, such as selecting or deleting from, updating, or inserting rows into base tables, views, or derived tables, the system aborts the request and returns an error to the requestor.

These rules apply equally to a join index defined with a partitioned primary index and to a column-partitioned join index whether the partitioning is single-level or multilevel.

Object Identifier Columns

For each BLOB, CLOB, or XML column created for a table, Vantage automatically stores a 40-byte or 45-byte multiple field pointer in the row to the subtable that stores the actual BLOB, CLOB, or XML data for that column (see [Sizing a LOB or XML Subtable](#)). This value, referred to as an OID, is stored in VARBYTE format.

If a BLOB, CLOB, or XML column is null, then so is its OID.

The column sizes reported here are for OIDs stored on disk in rows. They do not include, and are not the same as, the sizes of OIDs passed as inline or deferred host parameters as part of a USING request modifier row.

System-Derived and System-Generated Column Data Types

The following table lists the data types for several system-derived and system-generated column data types:

Derived or Generated Column	Data Type	Default Title
OID	VARBYTE	None
ROWID	<ul style="list-style-type: none"> • BYTE(10) for 2-byte partitioning • BYTE(16) for 8-byte partitioning 	None. There is no default title for the ROWID keyword because you cannot specify it in the select list of any DML request. You can only specify ROWID in a CREATE JOIN INDEX DDL request.
Identity column	Any of the following. <ul style="list-style-type: none"> • BYTEINT • DECIMAL(<i>n</i>,0) The scale for a DECIMAL identity column must be 0. • INTEGER • NUMBER(<i>n</i>,0) 	Default title for the column designated as an identity column.

Derived or Generated Column	Data Type	Default Title
	<p>Only a fixed NUMBER type is permitted for identity columns, and its scale must be 0.</p> <ul style="list-style-type: none"> • NUMERIC(<i>n</i>,0) • SMALLINT • BIGINT <p>The upper limits for DECIMAL and NUMERIC types are the following.</p> <ul style="list-style-type: none"> • DECIMAL(18,0) • NUMERIC(18,0) <p>This is true even when the DBS Control flag MaxDecimal is set to 38 (see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143).</p> <p>You can define an identity column with more than 18 digits of precision, or even as a BIGINT or NUMBER(<i>n</i>,0) type, without the CREATE TABLE or ALTER TABLE request aborting, but the values generated by Vantage for the identity column remain limited to the DECIMAL(18,0) type and size.</p> <p>A table that is managed by Teradata® Unity™ cannot have an identity column. Teradata Unity instead uses its own mechanism to generate “identity column” values. For more information, see <i>Teradata® Unity™ User Guide</i>, B035-2520.</p>	
PARTITION	INTEGER	PARTITION
PARTITION#L <i>n</i>	<ul style="list-style-type: none"> • INTEGER for the 2-byte form of PPI. • BIGINT for the 8-byte form of PPI. 	PARTITION#L <i>n</i>

Data Type Considerations

Different column data types occupy different amounts of disk space. This topic examines the various Teradata data types and indicates their absolute sizes.

The information presented here is for sizing purposes only. For specific usage information about the various data types supported by Teradata, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Data Types And Hashing

The data types for the primary index column set also have an important effect on how rows hash. For example, a primary index value typed DECIMAL with one precision generally hashes to a different AMP than the same primary index value typed DECIMAL with a different precision.

Data Type Size Differences For Packed64 and Aligned Row Format Architectures

Several data types have different sizes depending on whether the rows of a system are formatted using a packed64 or an aligned row format. The row size increases for aligned row formats are not all due to increased data type sizes. Byte alignment issues also play a significant role in this increase. Furthermore, neither study examined the effects of the increased data block header size introduced by WAL on storage (see [Byte Alignment](#) for more information).

The following table lists the predefined data types that have different disk storage sizes for packed64 and aligned row formats. If a data type is not listed in the table, then its allocated size is identical for packed64 format and aligned row format systems.

Data Type	Packed64 Format Size (bytes)	Aligned Row Format Size (bytes)	Allocated Format Size (bytes)
TIME	6	4	8
TIMESTAMP	10	4	12
INTERVAL DAY TO SECOND	10	4	12
INTERVAL MINUTE TO SECOND	6	4	8
INTERVAL SECOND	6	4	8

Numeric Data Types

Differences Between Exact and Approximate Predefined Numeric Data Types

The ANSI/ISO SQL:2011 standard defines two families of predefined numeric data types.

- Exact
- Approximate

An exact predefined numeric data type is one that can represent a value exactly. Exact numeric types are divided between the true integer types, which specify a precision but not a scale, and the fractional types, which can specify both a precision and a scale.

Teradata also supports various forms of the NUMBER data type, which can be used to represent both exact numeric values and floating point numeric values, depending on the syntax used to define the value. By the definitions the ANSI/ISO SQL:2011 standard uses for exact and approximate predefined numeric data types, the NUMBER data type as implemented by Teradata is neither an exact nor an approximate numeric type.

Exact Numeric Data Types

The definition of an exact numeric data type used by the ANSI/ISO SQL:2011 standard states that to be an exact numeric, the type must have a precision and a scale expressed either in base 2 or base 10.

According to the ANSI/ISO SQL:2011 standard, the following types are members of the exact predefined numeric set.

The following types are members of the set of true integers.

- BIGINT
- INTEGER
- SMALLINT

The following types are members of the set of fractional exact numeric types.

- DECIMAL
- NUMERIC

Teradata also supports the following exact predefined numeric data type extensions to the ANSI/ISO SQL:2011 standard.

Data Type	Type Family
BYTEINT	true integer
NUMBER(<i>p</i>)	fractional exact numeric

Note:

You can use the NUMBER type to represent both fixed point and floating point values, depending on the syntax you use to specify the value.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the various exact numeric data types supported by Teradata.

Approximate Numeric Data Types

The following types are members of the ANSI/ISO SQL:2011 approximate predefined numeric set.

- DOUBLE PRECISION
- FLOAT
- REAL

Although FLOAT, REAL, and DOUBLE PRECISION are distinct types as defined by the ANSI/ISO SQL:2011 standard, Teradata treats them as if they are equivalent to one another. The approximate numeric data types represent floating point numbers.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the various approximate numeric data types supported by Teradata.

Floating Point NUMBER Types

The requirements of an exact NUMBER include that for any addition, subtraction, multiplication, and division operations, the result must be an exact numeric value.

The NUMBER(*p,s*) syntax is not exact because the results of the previously mentioned dyadic operations do not produce an exact numeric result value.

Teradata supports the following predefined floating point numeric data type extensions to the ANSI/ISO SQL:2011 standard.

Data Type	Reason the Type is Not Exact	Type Family
NUMBER(<i>p,s</i>)	The result of the dyadic operations +, -, *, and / on NUMBER(<i>p,s</i>) values do not produce an exact numeric value when they operate on the NUMBER(<i>p,s</i>) type.	fractional floating point numeric The exact floating point NUMBER types can be used anywhere an approximate numeric type is used, but NUMBER values are stored as exact values in the same way as the exact NUMBER types and with the same accuracy. Floating point NUMBER types are not approximate types.
NUMBER	NUMBER and NUMBER(*) do not specify a scale.	
NUMBER(*)		
NUMBER(*, <i>s</i>)	NUMBER(*, <i>s</i>) has a precision that cannot be expressed as an integer for either base 2 or base 10 arithmetic.	

Note:

You can use the NUMBER type to represent both fixed point and floating point values, depending on the syntax you use to specify the value.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the NUMBER data type.

Integer Data Types

BYTEINT: 1 Byte (All Platforms)	
Bit 0	Bits 1 - 7
Sign	-128 — +127

SMALLINT: 2 Bytes (All Platforms)

Bit 0	Bits 1 - 15
Sign	-32,768 — +32,767

INTEGER: 4 Bytes (All Platforms)

Bit 0	Bits 1 - 31
Sign	-2,147,483,648 — +2,147,483,647

BIGINT: 8 Bytes (All Platforms)

Bit 0	Bits 1 - 63
Sign	-9,223,372,036,854,775,808 — +9,223,372,036,854,775,807

The following table indicates the alignment requirements for each of the true integer numeric data types and their respective sizes on packed64 and aligned row format systems:

Data Type	Alignment	Packed64 Size (bytes)	Aligned Row Format\	Size (bytes)
BYTEINT	1	1	1	
SMALLINT	2	2	2	
INTEGER	4	4	4	
BIGINT	8	8	8	

Non-INTEGER Numeric Data Types

The following table indicates the alignment requirements for each of the non-true integer numeric data types and their respective sizes on platforms that use packed64 versus aligned row storage formats:

Data Type	Range of n	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Size for Aligned Row Format (bytes)
DECIMAL(n) NUMERIC(n)	$1 \leq n \leq 2$	1	1	1
	$3 \leq n \leq 4$	2	2	2
	$5 \leq n \leq 9$	4	4	4
	$10 \leq n \leq 18$	8	8	8

Data Type	Range of n	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Size for Aligned Row Format (bytes)
	$19 \leq n \leq 38$		4 Alignment is 4 because decimal numbers in this range are stored internally as four 32-bit integers on all systems.	16
NUMBER(n) NUMBER(n,s)	$0 \leq n \leq 38$ and 0	0 - 18	0 - 18	0 - 18
FLOAT REAL DOUBLE PRECISION	Not applicable	8	8	8
NUMBER NUMBER(*) NUMBER(*,s)	Not applicable	0 - 18	0 - 18	0 - 18

FLOAT/REAL/DOUBLE PRECISION: 8 Bytes (All Platforms)

Bit 0	Bits 1 - 10 Exponent	Bits 11 - 63 Mantissa
Sign	$2 * 10^{-307} \text{ — } 2 * 10^{308}$	

NUMBER: 0- 18 Bytes (All Platforms)

Byte 0	Bytes 1 - 17
Exponent	Mantissa

DECIMAL/NUMERIC [(n[,m])]: 1, 2, 4, 8, or 16 Bytes (All Platforms)

1 - 2 digits		Not used													
3 - 4 digits		Not used													
5 - 9 digits				Not used											
10 - 18 digits								Not used							
19-38 digits															
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
-99 — 99	Not used														
-9999 — 9999		Not used													
-999999999 — 999999999				Not used											
-9999999999999999 — 9999999999999999								Not used							
-99999999999999999999999999999999 — 99999999999999999999999999999999															

Byte Data Types

BYTE(n): 1 - 64 000 Bytes Fixed (n bytes) (All Platforms)

Byte 1	...	Byte 64,000
1 — 64,000		

VARBYTE(n): 1 - 64,000 Bytes Variable (\leq n bytes) (All Platforms)

Byte 1	...	Byte 64,000
1 — 64,000		

LONG VARBYTE(n): 1 - 64,000 Bytes Fixed (equivalent to VARBYTE (64000)) (All Platforms)

Byte 1	...	Byte 64,000
1 — 64,000		

BLOB(n): 8 bytes - 2,097,088,000 Bytes Variable (All Platforms)

Chunk	Bytes 1 - 8	...	Byte 64,000
1	length	1 — 64,000	
...	length	1 — 64,000	
m	length	1 — m	

The following table shows the alignment and size for byte types on both platforms.

Data Type	Packed64 Size	Aligned Row Size	Allocated Aligned Row Size
BYTE(n)	n	1	n
VARBYTE(n)	\leq n	\leq n	\leq n
LONGVARBYTE(n)	\leq n	\leq n	\leq n
BLOB(n)	n	1	n

DateTime Data Types

DATE: 4 Bytes (All Platforms)

1Year

1Month

Day

AD January 1, 1 — AD December 31, 9999

TIME: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)

Hour

Minute

Seconds

hh:mm:ss[.ssssss]

12 Bytes (Aligned Row Platforms) TIMESTAMP: 10 Bytes (Packed64 Platforms)

Year

Month

Day

Hour

Minute

Second

yyyy-mm-dd hh:mm:ss

TIME WITH TIME ZONE: 8 Bytes (All Platforms)

Hour

Minute

Second

Timezone_Hour

Timezone_Minute

hh:mm:ss.ssssss±hh:mm

TIMESTAMP WITH TIME ZONE: 12 Bytes (All Platforms)

Year

Month

Day

Hour

Minute

Second

Timezone_Hour

Timezone_Minute

yyyy-mm-dd hh:mm:ss±hh:mm

The following table indicates the alignment requirements for each of the ANSI/ISO DateTime data types and their respective sizes on packed64 and aligned row platforms. When the size of a stored value for a type differs for packed64 and aligned row platforms, the differing table cells are shaded.

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
DATE	4	4	4
TIME	6	4	8
TIMESTAMP	10	4	12
TIME WITH TIME ZONE	8	4	8
TIMESTAMP WITH TIME ZONE	12	4	12

Interval Data Types

INTERVAL YEAR: 2 Bytes (All Platforms)

Years Precision

1

2

3

4

Range

-9 — 9

Not used

-99 — 99 (default)

Not used

-999 — 999

Not used

-9999 — 9999

INTERVAL YEAR TO MONTH: 4 Bytes (All Platforms)

Years Precision

1

2

3

4

Months Precision

Range

Range

-9 — 9

Not used

-99 — 99

Not used

-999 — 999

Not used

-9999 — 9999

00 — 11

INTERVAL MONTH: 2 Bytes (All Platforms)

Months Precision

1

2

3

4

Range

-9 — 9

Not used

-99 — 99

Not used

-999 — 999

Not used

-9999 — 9999

INTERVAL DAY: 2 Bytes (All Platforms)			
Days Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99		Not used	
-999 — 999			Not used
-9999 — 9999			

INTERVAL DAY TO HOUR: 4 Bytes (All Platforms)					
Days Precision					Hours Precision
1	2	3	4		
Range					
-9 — 9	Not used				
-99 — 99 (default)		Not used			
-999 — 999			Not used		
-9999 — 9999					00 — 23

INTERVAL DAY TO MINUTE: 8 Bytes, including 2 pad bytes (All Platforms)				
Days Precision				Minutes Precision
1	2	3	4	
Range				
-9 — 9	Not used			00 — 59
-99 — 99 (default)		Not used		
-999 — 999			Not used	
-9999 — 9999				

INTERVAL DAY TO SECOND: 10 Bytes (Packed64 Platforms) 12 Bytes (Aligned Row Platforms)

Days Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			00.0 — 59.9	Not used				
-99 — 99 (default)		Not used		00.00 — 59.99		Not used			
-999 — 999			Not used	00.000 — 59.999			Not used		
-9999 — 9999				00.0000 — 59.9999				Not used	
				00.00000 — 59.99999					Not used
				00.000000 — 59.999999 (default)					

INTERVAL HOUR: 2 Bytes (All Platforms)

Hours Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99 (default)		Not used	
-999 — 999			Not used
-9999 — 9999			

INTERVAL HOUR TO MINUTE: 4 Bytes (All Platforms)

Hours Precision				Minutes Precision
1	2	3	4	
Range				
-9 — 9	Not used			
-99 — 99 (default)		Not used		00 — 59
-999 — 999			Not used	
-9999 — 9999				

INTERVAL HOUR TO SECOND: 8 Bytes (All Platforms)

Hours Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			00.0 — 59.9	Not used				
-99 — 99 (default)		Not used		00.00 — 59.99		Not used			
-999 — 999			Not used	00.000 — 59.999			Not used		
-9999 — 9999				00.0000 — 59.9999				Not used	
				00.00000 — 59.99999					Not used
				00.000000 — 59.999999					

INTERVAL MINUTE: 2 Bytes (All Platforms)

Minutes Precision			
1	2	3	4
Range			
-9 — 9	Not used		
-99 — 99		Not used	
-999 — 999			Not used
-9999 — 9999			

**INTERVAL MINUTE TO SECOND: 6 Bytes (Packed64 Platforms)
8 Bytes (Aligned Row Platforms)**

Minutes Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			0.0 — 59.9	Not used				
-99 — 99 (default)		Not used		0.00 — 59.99		Not used			
-999 — 999			Not used	0.000 — 59.999			Not used		
-9999 — 9999				0.0000 — 59.9999				Not used	
				0.00000 — 59.99999					Not used

INTERVAL MINUTE TO SECOND: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)									
Minutes Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
				0.000000 — 59.999999					

INTERVAL SECOND: 6 Bytes (Packed64 Platforms) 8 Bytes (Aligned Row Platforms)									
Seconds Precision				Fractional Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range				Range					
-9 — 9	Not used			0.0 — 0.9	Not used				
-99 — 99 (default)		Not used		0.00 — 0.99		Not used			
-999 — 999			Not used	0.000 — 0.999			Not used		
-9999 — 9999				0.0000 — 0.9999				Not used	
				0.00000 — 0.99999					Not used
				0.000000 — 0.999999 (default)					

This table shows the alignments for interval types and their respective sizes on both platforms. When the size of the stored value for a type differs for the platforms, the cells are shaded.

Data Type	Packed 64 Size (bytes)	64-Bit Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
INTERVAL YEAR	2	2	2
INTERVAL YEAR TO MONTH	4	2	4
INTERVAL MONTH	2	2	2
INTERVAL DAY	2	2	2
INTERVAL DAY TO HOUR	4	2	4
INTERVAL DAY TO MINUTE	8	2	8
INTERVAL DAY TO SECOND	10	4	12
INTERVAL HOUR	2	2	2
INTERVAL HOUR TO MINUTE	4	2	4

Data Type	Packed 64 Size (bytes)	64-Bit Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
INTERVAL HOUR TO SECOND	8	4	8
INTERVAL MINUTE	2	2	2
INTERVAL MINUTE TO SECOND	6	4	8
INTERVAL SECOND	6	4	8

Period Data Types

The following table indicates the alignment requirements for each of the Period data types and their respective sizes on aligned row format platforms. When the size of the stored value for a type differs for packed64 and aligned row format platforms, the differing table cells are shaded.

Data Type	Packed 64 Size (bytes)	64-Bit Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
PERIOD (DATE)	8	2	8
PERIOD (TIME(n))	12	2	16
PERIOD (TIME(n) WITH TIME ZONE)	16	2	16
PERIOD (TIMESTAMP(n))	20	2	24
PERIOD (TIMESTAMP(n) WITH TIME ZONE)	24	2	24

PERIOD (DATE): 8 Bytes (All Platforms)

Beginning Date	Ending Date
AD January 1, 1 — AD December 30, 9999	AD January 2, 1 — AD December 31, 9999

PERIOD (TIME (Precision)): 12 Bytes (Packed64 Platforms) 16 Bytes (Aligned Row Platforms)

Beginning Time (6 bytes - Packed64 Platforms) 8 bytes - Aligned Row Platforms			Ending Time (6 bytes - Packed64 Platforms) 8 bytes - Aligned Row Platforms		
Hour	Minute	Seconds	Hour	Minute	Seconds
hh:mi:ss[.ssssss]			hh:mi:ss[.ssssss]		

PERIOD (TIME (Precision) WITH TIME ZONE): 16 Bytes (All Platforms)**Beginning Time With Time Zone (8 bytes)****Ending Time With Time Zone (8 bytes)**

Hour	Minute	Second	Timezone _Hour	Timezone _Minute	Hour	Minute	Second	Timezone _Hour	Timezone _Minute
hh:mi:ss.ssssss±hh:mi					hh:mi:ss.ssssss±hh:mi				

PERIOD (TIMESTAMP (Precision)): 20 Bytes (All Platforms)**Beginning Time Stamp (10 bytes)****Ending Time Stamp (10 bytes)**

Year	Month	Day	Hour	Minute	Second	Year	Month	Day	Hour	Minute	Second
yyy-mm-dd hh:mi:ss						yyy-mm-dd hh:mi:ss					

PERIOD (TIMESTAMP (Precision)) When Ending Element Value is UNTIL_CHANGED: 11 Bytes (All Platforms)**Beginning Time Stamp (10 bytes)****Ending Time
Stamp
(1 byte)**

Year	Month	Day	Hour	Minute	Second	UNTIL_ CHANGED
yyy-mm-dd hh:mi:ss						uc

PERIOD (TIMESTAMP (Precision) WITH TIME ZONE): 24 Bytes (All Platforms)															
Beginning Time Stamp (10 bytes)								Ending Time Stamp (10 bytes)							
Year	Month	Day	Hour	Minute	Sec.	Time-zone _Hour	Time-zone _Minute	Year	Month	Day	Hour	Minute	Sec.	Time-zone _Hour	Time-zone _Minute
yyyy-mm-dd hh:mi:ss±hh:mi								yyyy-mm-dd hh:mi:ss±hh:mi							

PERIOD (TIMESTAMP (Precision)) When Ending Element Value is UNTIL_CHANGED: 13 Bytes (All Platforms)

Beginning Time Stamp With Time Zone (12 bytes)

Ending Time Stamp With Time Zone (1 byte)

Year	Month	Day	Hour	Minute	Second	Timezone_Hour	Timezone_Minute	UNTIL_CHANGED
------	-------	-----	------	--------	--------	---------------	-----------------	---------------

yyy-mm-dd hh:mi:ss±hh:mi

uc

Character Data Types

CHARACTER(n): 1 - 64,000 Bytes Fixed (All Platforms)

Byte 1

...

Byte 64,000

1 — 64,000

VARCHAR(n): 1 - 64 000 Bytes Variable (All Platforms)

Byte 1

Byte 2

...

Byte 64,000

Column offset

1 — 64,000

LONG VARCHAR: 1 - 64,000 Bytes Fixed (equivalent to VARCHAR(64000) (All Platforms)

Byte 1

Byte 2

...

Byte 64,000

Column offset

1 — 64,000

CLOB(n): 8 bytes - 2 097 088 000 bytes Variable (All Platforms)

Chunk

Bytes 1 - 8

...

Byte 64,000

1

length

1 — 64,000

...

length

1 — 64,000

m

length

1 — n
If a CLOB is composed of Unicode characters, its upper limit is 1,048,544,000 bytes.
In both cases, the maximum supported size is slightly less than 2 GB or 1 GB, respectively.

CHARACTER(n) CHARACTER SET GRAPHIC: 1 - 32,000 Bytes Fixed (All Platforms)

Byte 1

...

Byte 32,000

1 — 32,000

VARCHAR(n) CHARACTER SET GRAPHIC: 1 - 32,000 Bytes Variable (All Platforms)

Byte 1

Byte 2

...

Byte 32,000

Column Offset

1 — 32,000

LONG VARCHAR CHARACTER SET GRAPHIC: 32,000 Bytes Fixed (equivalent to VARCHAR(n) CHARACTER SET GRAPHIC (32000) (All Platforms)

Byte 1

Byte 2

...

Byte 32,000

Column Offset

1 — 32,000

The following table indicates the alignment requirements for the character data types and their respective sizes on packed64 and aligned row format platforms.

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
CHARACTER(n) CHARACTER SET LATIN	n	1	n
CHARACTER(n) CHARACTER SET KANJI			
CHARACTER(n) CHARACTER SET UNICODE	$2n$	2	$2n$
CHARACTER(n) CHARACTER SET GRAPHIC			
VARCHAR(n) CHARACTER SET LATIN	n	1	n
VARCHAR(n) CHARACTER SET UNICODE	$\leq 2n$	1	$\leq 2n$
VARCHAR(n) CHARACTER SET GRAPHIC			
VARCHAR(n) CHARACTER SET KANJI	$\leq n$	1	n
LONG VARCHAR CHARACTER SET LATIN	64,000	1	64,000
LONG VARCHAR CHARACTER SET KANJI1			

Data Type	Packed64 Size (bytes)	Aligned Row Size (bytes)	Allocated Aligned Row Size (bytes)
LONG VARCHAR CHARACTER SET UNICODE	32,000		32,000
LONG VARCHAR CHARACTER SET GRAPHIC			
LONG VARCHAR CHARACTER SET KANJISJIS			
CLOB(<i>n</i>) CHARACTER SET LATIN	<i>n</i>	1	<i>n</i>
CLOB(<i>n</i>) CHARACTER SET UNICODE	$\leq 2n$	2	$\leq 2n$

XML/XMLTYPE Data Type

Syntax	Inline Storage	Maximum Length	LOB/Non-LOB
XML	4,046 bytes	2,097,088,000 bytes	LOB
XML(<i>n</i>) where $n \leq 64,000$	<i>n</i> bytes	<i>n</i> bytes	Non-LOB
XML(<i>n</i>) where $n > 64,000$	4,046 bytes	<i>n</i> bytes	LOB
XML(<i>n</i>) INLINE LENGTH <i>m</i>	<i>m</i> bytes	<i>n</i> bytes	If $n \leq m$, Non-LOB Otherwise, LOB

XML data can be stored inline (inside a row) or outside the row in a LOB subtable or both, depending on the syntax used to specify it.

XML and XMLTYPE are synonyms.

For more information on the XML data type, see *Teradata Vantage™ - XML Data Type*, B035-1140.

JSON Data Type (Text-Based Format)

Note:

The text-based format is the default storage format for JSON data. Other storage formats for JSON data include binary and universal binary formats, as described in the following pages.

JSON data can be stored inline (inside a row) or outside the row in a LOB subtable or both, depending on the syntax used to specify it:

Syntax	Default Inline Storage	Maximum Length	LOB/Non-LOB
JSON	4,046 bytes	16,776,192 bytes	LOB
JSON(<i>n</i>) CHARACTER SET LATIN, where <i>n</i> ≤ 64,000	<i>n</i> characters	<i>n</i> characters	Non-LOB
JSON(<i>n</i>) CHARACTER SET LATIN, where <i>n</i> > 64,000	4,096 bytes	<i>n</i> characters	LOB
JSON(<i>n</i>) CHARACTER SET LATIN INLINE LENGTH <i>m</i>	<i>m</i> characters	<i>n</i> characters	If <i>n</i> ≤ <i>m</i> , Non-LOB Otherwise, LOB
JSON(<i>n</i>) CHARACTER SET UNICODE, where <i>n</i> ≤ 32,000	<i>n</i> characters	<i>n</i> characters	Non-LOB
JSON(<i>n</i>) CHARACTER SET UNICODE, where <i>n</i> > 32,000	4,096 bytes	<i>n</i> characters	LOB
JSON(<i>n</i>) CHARACTER SET UNICODE INLINE LENGTH <i>m</i>	<i>m</i> characters	<i>n</i> characters	If <i>n</i> ≤ <i>m</i> , Non-LOB Otherwise, LOB

If there is no column length specified, the default length is the maximum.

For more information on the JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150.

JSON Data Type (Binary Format)

The JSON data type can be stored in binary format (BSON). Specify BSON with the optional STORAGE FORMAT clause in the data type specification syntax.

BSON data can be stored inline (inside a row) or outside the row in a LOB subtable or both, depending on the syntax used to specify it:

Syntax	Default Inline Storage	Maximum Length	LOB/Non-LOB
JSON(<i>n</i>) STORAGE FORMAT BSON, where <i>n</i> ≤ 64,000	<i>n</i> bytes	<i>n</i> bytes	Non-LOB
JSON(<i>n</i>) STORAGE FORMAT BSON, where <i>n</i> > 64,000	4,096 bytes	<i>n</i> bytes	LOB
JSON(<i>n</i>) STORAGE FORMAT BSON INLINE LENGTH <i>m</i>	<i>m</i> bytes	<i>n</i> bytes	If <i>n</i> ≤ <i>m</i> , Non-LOB Otherwise, LOB

If there is no column length specified, the default length is the maximum. The BSON type has a maximum total size of 16 MB (16,776,192 bytes).

Note:

The maximum size for BSON reflects the number of bytes in the data. This is different than the case for JSON text, where the maximum size reflects the number of characters in the data.

For more information on the JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150.

JSON Data Type (Universal Binary Format)

The JSON data type can be stored in universal binary (UBJSON) format. Specify UBJSON with the optional STORAGE FORMAT clause in the data type specification syntax.

UBJSON data can be stored inline (inside a row) or outside the row in a LOB subtable or both, depending on the syntax used to specify it:

Syntax	Default Inline Storage	Maximum Length	LOB/Non-LOB
JSON(n) STORAGE FORMAT UBJSON, where $n \leq 64,000$	n bytes	n bytes	Non-LOB
JSON(n) STORAGE FORMAT UBJSON, where $n > 64,000$	4,096 bytes	n bytes	LOB
JSON(n) STORAGE FORMAT UBJSON INLINE LENGTH m	m bytes	n bytes	If $n \leq m$, Non-LOB Otherwise, LOB

If there is no column length specified, the default length is the maximum. The maximum total size is 16 MB (16,776,192 bytes).

Note:

The maximum size for UBJSON reflects the number of bytes in the data. This is different than the case for JSON text, where the maximum size reflects the number of characters in the data.

For more information about the JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150.

DATASET Data Type

The DATASET data type includes a schema and data, which can both have a variable length. You can use the INLINE LENGTH option to specify an inline storage size. When the data is smaller than or equal to the inline storage size, it is stored inside the base row. Otherwise, the data is stored as a LOB (large object).

If the data is stored inline, it is treated as a non-LOB type. In this case, the performance may be better because there is no LOB overhead. You may see some performance improvement, especially when the data type is used with UDFs.

Each specification of the DATASET data type includes the following information:

- Maximum length
- In-line length
- Storage format
- Character set (comma-separated value (CSV) storage format only)
- Schema

Specify the `STORAGE FORMAT` option in the data type specification syntax. Available storage formats include Avro and CSV. The following values apply to either the schema or the data for the `DATASET` data type:

Storage Location	Maximum Length	Minimum Length	Default Length
LOB	16 MB	100 bytes	16 MB
Inline	64 KB	100 bytes	10 KB

[Optional] Specify a character set for the CSV format. It can be either `LATIN` or `UNICODE`. The default is the session character set.

A schema is optional for the CSV format. You can specify a schema in any supported JSON format. For instance-level `DATASET` values, the schema is stored in the same character set as the CSV data type. For column-level `DATASET` values, it is encoded in UTF-8. The schema is null-terminated.

The CSV storage format will support extensions to the CSV standard, such as user-specified column and record delimiters and header field names. If you use any of these extensions, specify a schema. You can define schemas at the column-level or instance-level for any of the built-in storage formats of the `DATASET` type. Column-level schemas are binding for all instances of the data type loaded into that particular column, but instance-level schemas may vary from instance to instance.

User-Defined Data Types

The required byte alignment and storage size of a UDT depends on how it is defined.

FOR this category of UDT ...	THE byte alignment and storage size are ...
Distinct	the same as that of the predefined data type on which it is defined.
Structured	different, depending on whether they are defined on the following type of platform: <ul style="list-style-type: none"> • Packed64 format systems • Aligned row format systems See Sizing Structured UDT Columns for information about how to determine the storage size of a given UDT column.

Array Data Types

The storage size of an `ARRAY`/`VARRAY` depends on how it is defined.

Field	Description	Size (bytes)
Flag bits	Used to configure the layout of the field. Bit 0: variable offset size. • If 0, the variable offsets are 2 bytes each. • If 1, the variable offsets are 4 bytes each. Bits 1 - 15: unused	2
Last present element	Index of the last present element in the array. This consists of 1 to <i>number_of_dimensions</i> integer values, one for each dimension. The index values are 0-based. If the array has uninitialized elements, the index values are equal to -1.	4 x number_of_dimensions
Presence bit array	Presence bit array for the array elements. A variable length byte array with 1 bit per element laid out in row major order. 0 means the element is null. 1 means the element is not null.	$\frac{\text{number_of_elements} + 7}{8}$
Variable offset array	<ul style="list-style-type: none"> If the element type is a variable length type, an offset array is used to index into the element data. The offset array is the same as the offset array in the table row. If the element type is fixed length, the variable offset array is omitted. 	2 x number_of_elements or 4 x number_of_elements
Element value array	All elements are stored in row major order. Variable length elements are stored in their actual size.	number_of_elements x element_size

Row Size Calculation

This topic describes a procedure for calculating the physical size of a row for a typical non-column-partitioned table stored on a system with a packed64 format architecture. The procedure does not account for BLOB, CLOB, or XML data, which are stored in 64 KB pieces in a subtable outside the base table row (see [Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables](#) and [Sizing a LOB or XML Subtable](#)), nor does it account for spool rows, which can be approximately 1MB long.

Use the Row Size Calculation form (see [Sample Worksheet Forms](#)) to record your calculations.

The following employee table row definition is used as a sample for this procedure:

Employee

Employee								
EmpNum	SupEmpNum	DeptNum	JobCode	LName	FName	HireDate	BDate	SalAmt
PK, SA	FK	FK	FK	NN	NN	NN	NN	NN
INTEGER	INTEGER	INTEGER	SMALLINT	CHAR(20)	VARCHAR(30)	DATE	DATE	DECIMAL(10, 2)

Procedure for Packed64 Systems

Note the following points about sizing UDT columns.

- The size of a distinct UDT column is identical to the size of the underlying predefined data type for the UDT.
- The size of a structured UDT column is more difficult to determine and is not equivalent to the sums of the sizes of its individual underlying predefined data types. The calculation is further complicated by the fact that structured types can be nested to a maximum of 512 levels.

See [Sizing Structured UDT Columns](#) for details about the composition and sizing of structured UDTs.

determine the sizes of each individual structured UDT

This table used for this example has no LOB or UDT columns, so you would skip steps 14, 15, and 16 of the procedure, jumping from step 13 directly to step 17.

Follow this procedure to determine the physical size of a typical row for any given non-LOB table:

1. List all the varying length columns in the four columns labeled Variable Data Detail.
2. For each variable length column, estimate the average number of bytes expected.
3. Add the total number of bytes in varying length columns and record the figure in the column labeled SUM(a).

In the example table, there is only one varying length column, FName, which is typed VARCHAR(30). Its average number of bytes is estimated to be 14, so the value for SUM(a) is 14.

4. Determine how many columns in the table there are for each fixed byte data type.

The example table has the following number of each fixed byte data type:

Data Type	Number of Columns
DATE	2
DECIMAL	1
INTEGER	3
SMALLINT	1

5. Enter these figures in the Number of Columns column next to each relevant data type.

Remember that all row lengths must be an even number of bytes (see [Byte Alignment](#)), so be sure to take this into account.

6. Multiply the counts by the sizing factor provided for the type and enter the results under the Total column.
7. Enter the total byte counts for the fixed length character types in the SUM(n) column.
 - CHARACTER
 - BYTE
 - GRAPHIC

For the example, only one CHARACTER column with a byte count of 20 is found, so enter 20 as the result for SUM(n).

8. Add the values for SUM(n) and SUM(a) and record them in the column labeled Logical Size.

For the example, the logical size is 64 bytes.

9. The overhead for any nonpartitioned primary index row is 14 bytes. For a PPI row, the overhead is 18 bytes. Those numbers are prerecorded for you in the column labeled Overhead. Use the appropriate column for the primary index type of the table.
10. Multiply the number of variable length columns by 2 to account for the 2-byte variable column offset pointers determined in step 1 of this procedure.

Write the value in the column labeled Variable Column Offsets.

For the example, there is only one variable column, so write the number 2 here ($1 \times 2 = 2$).

11. Record the number of columns compressed on a non-null value.
12. Record the number of nullable columns.
13. Divide the sum of step 11 and step 12 by 8 and record the quotient of the operation.

For aligned row format systems, you will use this value again at step 20a.

The purpose of this step is to account for any required additional presence bits that might be required.

For the example, the calculation is $3/8$, so the quotient is 0.

14. Determine how many BLOB, CLOB, or XML columns are in the table and record the number under Number of Columns on the Row Size Calculation Form, Page 1 of 2. Multiply that number by 40 to determine the total row size taken up by BLOB, CLOB, and XML object IDs (OIDs).
15. Compute and record the sizes of any UDT columns on page 2 of 2 of the Row Size Calculation Form as follows:
 - a. Column 1 records the name of the UDT being recorded.
 - b. Column 2 records how many columns in the table have that type.
 - c. Column 3 records the sizing factor for the UDT.

This is the physical size of the column, which is one of the following.

FOR this UDT type ...	The physical size is the ...
distinct	size of its underlying predefined data type.
structured	calculated size of the column.
ARRAY/VARRAY	<ul style="list-style-type: none"> size of its underlying predefined data type if it is a one-dimensional array. calculated size of the column if it is a multidimensional array.

- d. Column 4 records the product of the number of columns and the sizing factor for the UDT.
16. Sum the UDT totals and record them on Page 1 of 2 of the Row Size Calculation Form in the cell labeled UDTs.

17. Add the integers recorded in the Total column and record the sum in the column labeled Physical Size.
18. If the sum is an uneven number, round it up to the next even number.

For the example, the sum is 82, so no rounding is necessary.

19. Whether you continue or not depends on the addressing used by your system.

IF you are calculating the row size for this type of system ...	THEN ...
packed64	stop.
aligned row	continue to step 20.

20. Determine the byte alignment overhead for the row.

- a. Set the value of *64-bit_byte_alignment_bit_overhead* to 0 and consult the number of additional presence bits determined in Step 13 of this procedure.

IF the value recorded in Step 13 is an ...	THEN set 64-Bit_Byte_Alignment_Bit_Overhead to this value ...
odd number AND the number of variable length columns in the row is ≥ 1	1
even number	0

- b. Determine the maximum alignment among all the fixed length columns in the row.
Call this variable *FA*.
- c. Determine the maximum alignment among all compressible columns in the row.
Call this variable *CA*.
- d. If there are no fixed length or value compressible columns in the row, set the value of *FA* or *CA* to 1.

21. Increment the value of *Byte_Alignment_Bit_Overhead* by $\text{MAX}(FA, CA) - 1$.

22. Determine the maximum alignment of all variable length columns.

Call this variable *VA*.

If there are no variable length columns, set *VA* to 1.

23. Increment the value of *64-Bit_Byte_Alignment_Bit_Overhead* by $(VA - 1)$
24. Add the value of *64-Bit_Byte_Alignment_Bit_Overhead* to the aligned row size determined by Steps 14 - 17 in this procedure.
25. Round up the value determined in Step 24 to the nearest multiple of 8.

This value is the row size in bytes for an aligned row format system.

Note that steps 20 - 25 are a conservative approximation of the row size for an aligned row format system.

Procedure To Determine the Exact Row Size for Aligned Row Systems

The following procedure determines the exact row size for aligned row format systems.

1. Set the initial value for *RowSize* to 12 bytes and the initial value for *64-Bit_Byte_Alignment_Bit_Overhead* to 0 bytes.
2. Record the number of columns compressed on a non-null value.
3. Record the number of nullable columns.
4. Divide the sum of step 2 and step 3 by 8 and record the quotient of the operation.

Call this variable *PB*.

The purpose of this step is to account for any required additional presence bits that might be required.

For example, if the ratio is 3/8, then the quotient is 0.

5. Overwrite the value for *RowSize* as follows:

$$\text{RowSize} = \left(\frac{(\text{PB} + 7)}{8} \right) - 1$$

6. Record the number of variable length columns.

Call this variable *VC*.

7. Determine the space required for the variable length columns offset array using the following pseudocode procedure:

```

IF VC > 0
  THEN
    IF (RowSize is odd)
      THEN
        RowSize += 1
        64BitOverhead += 1
      ENDIF
    RowSize = RowSize + 2 * (VC + 1)
  END IF

```

where += is the assignment operator.

8. Determine the maximum alignment required for all fixed length columns.
Call this variable *FA*.
9. Determine the size of all fixed length columns.
Call this variable *FS*.
10. Determine the maximum alignment required for all compressible columns.
Call this variable *CA*.

11. Determine the size of all compressible columns.

Call this variable *CS*.

12. Determine the maximum alignment required for all variable length columns.

This includes BLOB, CLOB, and XML columns, VARCHAR columns, and VARBYTE columns. Because BLOB, CLOB and XML values are stored in subtables outside of the row, this calculation should more correctly be referred to as a table size determination rather than a row size determination.

Call this variable *VA*.

13. Determine the size of all variable length columns.

Call this variable *VS*.

14. Calculate the actual size value (call it *FixedActual*) for fixed length columns using the following equation, where *FP* represents row size:

$$\text{FixedActual} = (\text{FP} + \text{FA} - 1) - (\text{FP} + \text{FA} - 1) \text{ MOD}(\text{FA})$$

The modulo(*FA*) adjustment aligns the value of *FixedActual* to the nearest multiple of *FA*.

15. Calculate the fixed length column overhead (call it *B*) using the following equation:

$$B = \text{FixedActual} - \text{FP}$$

16. Set *RowSize* = *FixedActual* + *FS*.

Call this variable *CP*.

17. Calculate the actual size value (call it *CompressActual*) for compressible length columns using the following equation:

$$\text{CompressActual} = (\text{CP} + \text{CA} - 1) - (\text{CP} + \text{CA} - 1) \text{ MOD}(\text{CA})$$

The modulo(*CA*) adjustment aligns the value of *CompressActual* to the nearest multiple of *CA*.

18. Calculate the value compressible column overhead (call it *C*) using the following equation:

$$C = \text{CompressActual} - \text{CP}$$

19. Set *RowSize* = *CompressActual* + *CS*.

20. Set *64-Bit_Byte_Alignment_Bit_Overhead* = *64-Bit_Byte_Alignment_Bit_Overhead* + *B* + *C*.

21. Adjust the values of *RowSize* and *64-Bit_Byte_Alignment_Bit_Overhead* if *B* + *C* > MAX(*FA*, *CA*) using the following pseudocode procedure:

```
IF ((B+C) > MAX(FA,CA))
  THEN
    RowSize = RowSize - CA
    Byte_Alignment_Bit_Overhead = Byte_Alignment_Bit_Overhead - CA
  END IF
```

22. Assign *RowSize* to *VP*.

23. Calculate the actual maximum alignment required for variable length columns using the following equation:

$$VA_Actual = (VP+VA-1)-(VP+VA-1)MOD(VA)$$

The modulo(VA) adjustment aligns the value of *VA_Actual* to the nearest multiple of VA.

24. Set *Byte_Alignment_Bit_Overhead* = *Byte_Alignment_Bit_Overhead* + *VA_Actual* - *VP*.

25. Set *RowSize* = *VA_Actual* + *VS*.

26. Calculate the actual value for *RowSize* using the following equation:

$$RowSize = (RowSize+7) - (RowSize + 7)MOD(8)$$

The modulo(8) adjustment rounds the value for *RowSize* upward to the nearest multiple of 8.

27. Calculate the actual value for *64-Bit_Byte_Alignment_Bit_Overhead* using the following equation:

$$64\text{-}Bit_Byte_Alignment_Bit_Overhead = 64_Bit_Overhead + (RowSize + 7) - (RowSize + 7)MOD(8)$$

The modulo(8) adjustment rounds the value for *64-Bit_Byte_Alignment_Bit_Overhead* upward to the nearest multiple of 8.

28. Calculate the total aligned row format size by adding the values of *RowSize* and *64-Bit_Byte_Alignment_Bit_Overhead* using the following equation:

$$64\text{-}Bit_Byte_Alignment_Bit_Overhead = RowSize + 64 - 64\text{-}Bit_Byte_Alignment_Bit_Overhead$$

Sizing Databases, Users, and Profiles

Underestimating space requirements for databases, users, and profiles leads to performance problems. This is because Vantage requires free disk cylinders to enable the growth of permanent, temporary, and spool space as well as to enable the growth of permanent journal tables.

These tables cannot share cylinders when you make your permanent, temporary, and spool disk space assignments. For databases with uneven or skewed data distribution, current space assignments may be much higher than normal to accommodate the skewed use. In such cases, global space accounting can help and space assignments can be normal (for example, an average value) because space is managed at a global level. For more information about global space accounting, see [About Global Space Accounting](#).

The best practice for the initial sizing of the disk space required by a database, user, or profile is to make a good estimate of the space these will require when a database, user, or profile is created, and then to modify those assignments at a later time using MODIFY DATABASE, MODIFY USER, or MODIFY PROFILE requests as appropriate.

You should always consider the following disk space requirements when making disk space assignments for databases, users, and profiles.

Vantage requires this type of disk space ...	For ...
spool	materializing volatile tables.
temporary	materializing global temporary tables. To materialize global temporary tables, temporary space must have enough empty disk cylinders to contain their rows.

Modifying the Disk Space Currently Assigned to Databases, Users, or Profiles

Modifying the permanent, temporary, and spool space assignments for databases, user, and profiles is a simple operation using MODIFY DATABASE, MODIFY USER, and MODIFY PROFILE requests as appropriate.

You can use the system views listed here to help determine new values to for permanent, temporary, and spool space assignments. For more information about these views, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

- DBC.AllSpaceV[X]

Use this view to report disk space usage, including spool space, for any account, database, table, or user.

The following request reports how the space currently used by the department table is distributed on each AMP.

```
SELECT DatabaseName, TableName, AMP, CurrentPerm
FROM DBC.AllSpaceV
WHERE TableName = 'department'
ORDER BY 1,2,3;
```

DatabaseName	TableName	AMP	CurrentPerm
-----	-----	----	-----
test	department	1-0	1,024
test	department	1-1	512
test	department	1-2	1,024
test	department	1-3	512
personnel	department	1-0	2,048
personnel	department	1-1	1,536
personnel	department	1-2	1,536
personnel	department	1-3	1,536
user1	department	1-0	2,048
user1	department	1-1	1,536
user1	department	1-2	1,536
user1	department	1-3	1,536

The following request reports the values for the MaxPerm and CurrentPerm columns for each table contained by user. Because user only contains one table, employee, the request only returns information about employee and All, where the All “table” represents all of the tables contained by the specified database or user. In this case, All represents all of the tables contained by the user named user.

The MaxPerm value for All is the amount of permanent space defined for user. Because user contains only one table, the number of bytes on each AMP is the same for both All and employee. All of the reported values represent the size of the employee table.

Note that employee returns 0 bytes in the MaxPerm column because tables do not have MaxPerm space. Only databases and users have MaxPerm space, represented by All.

```
SELECT Vproc, TableName (FORMAT 'X(20)'), MaxPerm, CurrentPerm
FROM DBC.AllSpaceV
WHERE DatabaseName = user
ORDER BY TableName, Vproc;
```

Vproc	TableName	MaxPerm	CurrentPerm
-----	-----	-----	-----
0	All	2,621,440	64,000
1	All	2,621,440	64,000
2	All	2,621,440	112,640
3	All	2,621,440	112,640
...
0	employee	0	41,472
1	employee	0	41,472
2	employee	0	40,960
3	employee	0	40,960
...

- DBC.DiskSpaceV[X]

Use this view to report disk space usage, including pool space, for any account, database, or user.

The following request reports the permanent disk space across all AMPs.

```
SELECT AMP, DatabaseName, CurrentPerm, MaxPerm
FROM DBC.DiskSpaceV;
```

AMP	DatabaseName	CurrentPerm	MaxPerm
---	-----	-----	-----
.	.	.	.
.	.	.	.
0-0	stst14	0	125,000
0-0	ud12	0	125,000
1-0	atest	1,536	125,000
1-0	a1	0	247,500
1-0	btest	3,584	5,000
1-0	b2test	49,664	250,000
.	.	.	.
.	.	.	.
1-1	atest	1,536	125,000
1-1	a1	0	247,500
1-1	btest	3,584	5,000
1-1	b2test	50,688	250,000
.	.	.	.

.	.	.	.
1-2	atest	1,536	125,000

Similarly, you can submit a request like the following to calculate the percentage of space used by a particular database. Note that the request uses a NULLIFZERO specification to avoid a divide by zero exception.

```
SELECT DatabaseName, SUM(MaxPerm), SUM(CurrentPerm),
       ((SUM (CurrentPerm))/ NULLIFZERO (SUM(MaxPerm)) * 100)
       (FORMAT 'zz9.99%', TITLE 'Percent // Used')
FROM DBC.DiskSpaceV
GROUP BY 1
ORDER BY 4 DESC;
```

This request reports the following information from DBC.DiskSpaceV.

DataBaseName	Sum(MaxPerm)	Sum(CurrentPerm)	Percent Used
-----	-----	-----	-----
Finance	1,824,999,996	1,796,817,408	98.46%
DBC	12,000,000,006	8,887,606,400	73.98%
Spool_Reserve	2,067,640,026	321,806,848	15.56%
CrashDumps	300,000,000	38,161,408	12.72%
SystemFE	1,000,002	70,656	7.07%

- DBC.GlobalDBSpaceV[X][_SZ]

Use this view to see global space accounting information, including the following data:

- The space limits for a database or user at the system level
- The sizes of the total space allocated to AMPs in the system
- The total peak allocated space sizes across the system
- The skew limits associated with the different space values

The following request reports unallocated space remaining in the database:

```
SELECT databasename, allocatedperm, maxperm,maxperm-allocatedperm (TITLE
'UnallocatedSpace'), permskew
FROM DBC.GlobalDBSpaceV where databasename='example2';
```

This request reports the following information:

DatabaseName	example2
AllocatedPerm	5,148

```

MaxPerm          5,200
UnallocatedSpace 52
PermSkew  10,000

```

The maximum allocated peak spool for a given user:

```

SELECT peakallocatedspool, databasename
FROM DBC.GlobalDBSpaceV ORDER BY 1 DESC;

```

```

*** Query completed. 30 rows found. 2 columns returned.
*** Total elapsed time was 1 second.

```

PeakAllocatedSpool	DatabaseName
71,273,111,876	example
71,273,111,876	example2
0	LockLogShredder
0	TDMaps
0	tdwm
0	SysAdmin
0	Default
0	All
0	PUBLIC
0	SYSSPATIAL
0	SYSUIF
0	TDPUSER
0	TD_SYSXML
0	TD_SYSGPL
0	SYSLIB
0	External_AP
0	Crashdumps
0	SYSJDBC
0	TD_SERVER_DB
0	TDStats
0	SYSUDTLIB
0	SYSBAR
0	SystemFe
0	SQLJ
0	Sys_Calendar
0	TD_SYSFNLIB
0	TDQCD
0	EXTUSER

```
0 dbcmngr
0 DBC
```

- DBC.TableSizeV[X]

Use this view to report disk space usage excluding spool space, for any account or table.

The following request reports the total disk space currently used by the employee table with its peak space usage.

```
SELECT SUM(PeakPerm), SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE TableName = 'employee';
Sum(PeakPerm)    Sum(CurrentPerm)
-----
260,608          260,608
```

In this case, the 2 sums match, indicating that the disk space currently used by the employee table is also its peak space usage.

The following request reports poorly distributed tables by returning the CurrentPerm values for table_2 and table_2_nusi, the only tables contained by user across all AMPs.

```
SELECT Vproc, TableName (FORMAT 'X(20)'), CurrentPerm, PeakPerm
FROM DBC.TableSizeV
WHERE DatabaseName = user
ORDER BY TableName, Vproc;
Vproc    TableName                CurrentPerm    PeakPerm
-----
0        table_2                41,472         53,760
1        table_2                41,472         53,760
2        table_2                40,960         52,736
3        table_2                40,960         52,736
4        table_2                40,960         52,760
5        table_2                40,960         52,760
6        table_2                40,960         52,272
7        table_2                40,960         52,272
0        table_2_nusi          22,528         22,528
1        table_2_nusi          22,528         22,528
2        table_2_nusi          71,680         71,680
3        table_2_nusi          71,680         71,680
4        table_2_nusi          9,216          9,216
5        table_2_nusi          9,216          9,216
6        table_2_nusi          59,392         59,392
7        table_2_nusi          59,392         59,392
```

The report indicates that table_2 is evenly distributed across the AMPs, but that table_2_nusi is not, with its current permanent space clumped into spaces of 9,216 bytes, 22,528 bytes, 59,392 bytes, and 71,680 bytes across 2 AMPs each.

Different Disk Space Views Return Different Results

It is important to understand that because the DBC.AllSpaceV, DBC.DiskSpaceV, and DBC.TableSizeV system views report different information, using superficially identical requests to return information using those views often returns conflicting results. For example, while DBC.AllSpaceV and DBC.DiskSpaceV include information about spool space in their reports, DBC.TableSizeV does not.

Similarly, selecting SUM(CurrentPerm) from each of these views returns different results. For example, requesting SUM(CurrentPerm) from DBC.AllSpaceV returns a sum of All “tables,” representing the database or user total plus the sum of each table in the database or user. Submitting the same request using DBC.TableSizeV returns sums for all tables except for All, and submitting the query using DBC.DiskSpaceV returns sums only for All.

The following requests illustrate the different results the same request returns for the same information, but using different system views to access the data.

- DBC.AllSpaceV

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.AllSpaceV
WHERE DatabaseName = user;
Maximum(CurrentPerm)      Sum(CurrentPerm)
-----
112,640                    1,308,672
```

The reported values represent the disk space for all of the tables contained by user plus the disk space for the All table.

- DBC.DiskSpaceV

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.DiskSpaceV
WHERE DatabaseName = user;
Maximum(CurrentPerm)      Sum(CurrentPerm)
-----
112,640                    654,336
```

The reported values represent the disk space for the All table.

- DBC.TableSizeV

```
SELECT MAX(CurrentPerm), SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE DatabaseName = user;
```

Maximum(CurrentPerm)	Sum(CurrentPerm)
-----	-----
71,680	654,336

The reported values represent the disk space for all of the tables contained by user, but do not include the All table.

Use the following table to determine which system views are appropriate for requesting disk space information for different database objects.

To report disk space information at this level ...	You should use this system view ...
table	DBC.TableSizeV
database, user, or profile	DBC.DiskSpaceV
database or user with global space accounting	DBC.GlobalDBSpaceV
tables plus database, user, or profile	DBC.AllSpaceV

Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables

Vantage data block sizes and alignments are somewhat variable. This permits a designer the flexibility of designing tables without having to spend time balancing data types, row sizes, and block sizes in order to optimize their storage.

Critical Sizing Variables

The critical variables for sizing database tables and indexes are the following:

- Row size
- Estimated cardinality
- Presence of fallback

Table headers are a fixed variable, but must be considered when you are evaluating the projected sizes of your tables. Ensure that you take multivalue compression into account when sizing table headers (see [Compression Types Supported by Vantage](#)).

Sizing a Column-Partitioned Table

Column-partitioned tables have similar space usage as a table with row partitioning except for the impact of compression.

A column-partitioned table or join index may incur a large size increase compared with a table that is not column-partitioned if any of the following are true:

- There are few physical rows in populated combined partitions.
- There are many column partitions with ROW format and the subrows are narrow.

- There are few rows per unique value of the primary index (for a unique primary index, there would be only one row per unique value).

A container can only hold values of rows that have either the same internal partition number and hash bucket value (for PI or NoPI tables and join indexes) or hash value (for PI tables and join indexes). The potentially increased size is because of the increased number of physical rows and the overhead of the row header for each physical row. It is far more common for a column-partitioned table to be smaller, and often significantly so, than a table that is not column-partitioned if COLUMN format is used for narrow column partitions and the table is not over-partitioned because of the reduced number of physical rows and autocompression.

Use the following general procedure to size a column-partitioned table.

1. Estimate the size of the table without column partitioning using the standard methods for calculating the size of a table.
2. Adjust your estimate based on the expected impact of autocompression, row header compression, and row header expansion.

You can estimate the necessary adjustment by measuring the impact of your proposed column partitioning scheme on some sample data.

LOB and XML Sizing Variables

Each chunk of a BLOB, CLOB, or XML string stores an 8-byte length followed by as much as a 64 KB data fragment. LOBs are stored outside its base table row in a subtable. See [Sizing a LOB or XML Subtable](#) for more information.

Sizing Base Tables, Hash Indexes, and Join Indexes

Estimate the size of your base tables and hash and join indexes using the equations provided in the sections below.

These equations assume a constant block size. Because block sizes can vary widely within a table, you can obtain more finely tuned estimates by computing values over a range of block sizes.

Calculating a Typical Block Size

Depending on how you define your tables and the boundary conditions on their rows, the maximum size in bytes for a typical block can be estimated by the following equation:

Typical block size = $\text{ROUNDUP}(\text{MAX} \times \frac{3}{4}, 512)$

where:

This variable ...	Specifies ...
ROUNDUP	the ROUNDUP function.
$\text{MAX} \times \frac{3}{4}$	three quarters of the maximum block size in bytes. The upper limit on this value is 255 sectors, or 130,560 bytes.
512	the rounded up sector boundary, which is the number of bytes per sector for the table.

Sizing a Column-Partitioned Join Index

See [Sizing a Column-Partitioned Table](#) for information about sizing a column-partitioned join index. The methodology for column-partitioned join indexes is identical to that used to size a column-partitioned table.

Table Sizing Equations

The following parameter definitions are used with this equation set:

Parameter	Definition
Block Overhead	Block Header + Block Trailer = 72 bytes + 2 bytes = 74 bytes
Minimum Table Header Size	512 bytes
Typical Block Size	Number of bytes per block (see <i>Calculating a Typical Block Size</i> above).
NumAmps	Number of AMPs in the configuration.
RowCount	Estimated cardinality for the table.
Average RowSize	Physical row size for the table (see Row Size Calculation). Remember that all row lengths must be an even number of bytes (see Byte Alignment), so be sure to take this into account.

Rows per block (rounded down) = Typical block size – 38 ÷ Row size

Number of blocks (rounded up) = Row count ÷ Rows per block

Number of table header bytes = (Number of AMPs)(512)

Number of base table bytes (without fallback) = (Number of blocks × Typical block size) + Number of header bytes

Number of base table bytes (with fallback) = 2(Number of blocks)(Typical block size) – Number of header bytes

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

Example: Table, Hash, or Join Index on a Packed64 Format System

You have collected the following information.

Parameter	Value
Block size	5,120 bytes Note: Typical block size used.

Parameter	Value
Number of AMPs	20
Row count	1,000,000
Row size	100 bytes
Rows per block are rounded down. Number of blocks are rounded up.	

Calculate the size of this object when it is defined with fallback.

Number of rows per block = $(5,120 - 16) \div 100 = 51$

Number of blocks = $1,000,000 \div 51 = 19,608$

Number of table header bytes = $20 \times 512 = 10,240$

Number of bytes in base table with fallback = $((19,608 \times 5,120) \times 2) + 10,240$

Sizing a LOB or XML Subtable

One LOB or XML subtable is required for each column in a base table defined with a BLOB, CLOB, or XML data type. Depending on the size of each LOB or XML string, the cardinality of the subtable can exceed the cardinality of the base table. Estimate the size of your LOB and XML subtables using the equations provided in *LOB and XML Subtable Sizing Equation* below.

Because LOBs and XML strings are variable length entities, there are no alignment issues for aligned row format systems.

LOB and XML Subtable Rows and Skew

Each LOB or XML string is stored in 64 KB sections in a subtable, except for the row that stores the last section (because the number of bytes in a LOB or XML string is typically not an exact multiple of 64 KB). This means that for any LOB or XML string greater than 64 KB, there is not a 1:1 equivalence between the number of base table rows and the number of LOB or XML subtable rows supporting those base table rows. As a result, unless every LOB or XML string in a table has the identical size, it is likely that storage skew will occur.

IF the primary index for a base table is a ...	THEN ...
UPI	this skew does not affect performance in any way, but it is likely to produce an asymmetric storage effect on storage requirements.
NUPI	both of the following assertions are true: <ul style="list-style-type: none"> • Performance costs accrued by a skewed distribution of base table rows are likely to be further magnified. • Storage skew is even more likely to occur than for tables with a UPI.

LOB and XML Subtable Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
BaseRowCount	Estimated cardinality for the base table.
OIDSize	<ul style="list-style-type: none"> 40 bytes for nonpartitioned primary index and 2-byte PPI tables. 45 bytes for 8-byte PPI tables.
RowOverhead	The number of bytes devoted to subtable row overhead. <ul style="list-style-type: none"> If the base table has a non-partitioned primary index, the subtable row overhead is 12 bytes. If the base table has a partitioned primary index, the subtable row overhead is 16 bytes.
AvgLOBSize	Average LOB or XML string size for the table. Remember that all row lengths must be an even number of bytes (see Byte Alignment), so be sure to take this into account.

$$\text{LOB subtable size} = (\text{BaseRowCount}) \left(\text{OIDSize} \left((\text{RowOverhead} + 64000) \left(\frac{\text{AvgLOBSize}}{64000} \right) \right) \right)$$

LOB and XML Subtables and Fallback

The LOB or XML subtable size must be doubled if the base table for which the LOB or XML column is defined is fallback protected.

Sizing a Unique Secondary Index Subtable

Estimate the size of your unique secondary indexes using the equations provided below.

Because the index is unique, there is one USI subtable row for each row in the base table, so the row counts are identical.

USI Sizing Equation

The following parameter definitions are used with this equation:

Parameter	Definition
Index Value Size	Size of the column set on which the USI is defined. This must include any trailing pad bytes added for alignment purposes. All row lengths must be an even number of bytes (see Byte Alignment), so be sure to take this into account.
Block Overhead	Sum of the following factors.

Parameter	Definition
	<ul style="list-style-type: none"> • Block headers and trailers • Row headers and trailers • USI rowID • Spare byte • Presence bit arrays • Base table rowID = 28 bytes for a nonpartitioned primary index table but = 30 bytes for a PPI table

Packed64 Format Sizing Equations

NPPI USI subtable size = ((Row Count) × (Index Value Size + 28))

PPI USI subtable size = ((Row Count) × (Index Value Size + 30))

Aligned Row Format Sizing Equations

NPPI USI subtable size = ((Row Count) × (Index Value Size + 28))

PPI USI subtable size = ((Row Count) × (Index Value Size + 30))

If fallback is defined for the base table, then double the calculated result.

Vantage implicitly creates a unique secondary index on any column set specified as PRIMARY KEY or UNIQUE, so you must take these indexes into consideration for your capacity planning as well. PRIMARY KEY and UNIQUE constraints are generally implemented as single-table join indexes for temporal tables (see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details). You must also take any of these system-defined join indexes into account when doing capacity planning.

Sizing a Nonunique Secondary Index Subtable

Estimate the size of your nonunique secondary indexes using the equation provided below.

The number of base tables that can be referenced is limited by the maximum row size for the system and the length of the secondary index value. See below for more information about sizing NUSI subtables.

Special Considerations for NUSI Size Estimates

The number of AMPs in the configuration are an important factor in estimating the total size of any NUSIs defined on a base table, as summarized in the following table:.

IF the number of AMPS is ...	THEN at least ...	AND the result is that ...
less than the number of rows per value	one row from each NUSI value is probably distributed to each AMP.	<ul style="list-style-type: none"> • Every AMP has every value. • Every AMP has a subtable row for every value.

IF the number of AMPS is ...	THEN at least ...	AND the result is that ...
greater than the number of rows per value	some AMPs are missing some NUSI values.	<ul style="list-style-type: none"> • Not every AMP has every value. • Not every AMP has a subtable row for every value.

NUSI Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
Cardinality x 8 Cardinality x 10	<p>Each base table rowID is stored in a NUSI subtable.</p> <ul style="list-style-type: none"> • For a nonpartitioned primary index table, the row ID is 8 bytes long. • For a PPI table, the row ID is 10 bytes long. <p>This means that you must use the Cardinality * 8 factor for NUSI subtables for nonpartitioned primary index base tables and the Cardinality * 10 factor for NUSI subtables for PPI base tables.</p> <p>See equations below.</p>
NumDistinct	The value is an estimate of the number of distinct NUSI subtable values and is based on each NUSI subtable having at least one index row per AMP for each distinct index value of a base table row stored on that AMP.
IndexValueSize	The number of index data bytes.
NUSI Row Overhead	<p>Sum of the following factors.</p> <ul style="list-style-type: none"> • Row headers and trailers • NUSI row rowID • Spare byte • Presence octets = 18 bytes
MIN (NumAmps Rows per value)	The lesser of the two parameters.
6	The number of bytes consumed by the 3 VARCHAR Offset fields that follow the Additional Overhead field.

If fallback is defined for the base table, then double the calculated result.

NUSI Sizing Equation for Nonpartitioned Primary Index Base Table

NUSI subtable size $_{NPI} = 8 \times (\text{Cardinality}) + ((\text{NumDistinct}) \times (\text{IndexValueSize} + 18)) \times \text{MIN}(\text{NumAMPs} | \text{Rows per value})) + 6$

NUSI Sizing Equation for PPI Base Table

NUSI subtable size $_{PPI} = 10 \times (\text{Cardinality}) + ((\text{NumDistinct}) \times (\text{IndexValueSize} + 18)) \times \text{MIN}(\text{NumAMPs} | \text{Rows per value})) + 6$

NUSI Space Considerations

The PERM space required during the creation of a NUSI might temporarily be much greater than the space occupied by the finished index as described by the following table.

IF you create this sort of table ...	THEN the following peak temporary PERM space usage factors apply ...
non-fallback or fallback with small AMP cluster size	Estimate the temporary PERM space required when building a NUSI using the following worst case estimation equation. $\text{TemporarySpaceNF} = \text{Cardinality} \times (\text{LengthOfKey} + 30)$ where: <ul style="list-style-type: none"> • <i>TemporarySpaceNF</i> is the size of the temporary PERM space required without fallback. • <i>Cardinality</i> is the number of rows in the base table. • <i>LengthOfKey</i> is the combined byte length of the key.
fallback	Estimate the temporary PERM space required, assuming typical AMP cluster size, using the following $\text{TemporarySpaceFB} = \text{NUSISubtableSize} + \text{TemporarySpaceNF}$ This is a very conservative space estimate. For typical AMP cluster sizes, peak usage exceeds the prediction made by the model.

AMP cluster size, which determines the relative size of the backup subtables, is an important factor. For information about AMP clusters, see *Teradata Vantage™ - Database Introduction*, B035-1091. A typical AMP cluster size is 4 AMPs, but the valid range varies from 2 to 8 AMPs per cluster.

NUSIs do not use spool space and are built one subtable at a time.

Sizing User-Defined Routines

You must consider the size of any user-defined functions, methods, and procedures stored within the database when you are doing capacity planning for your system. This includes any of the UDFs you create to enforce algorithmic compression and row-level security privileges.

There are no special sizing considerations for these routines beyond planning for the space they occupy on your system.

Sizing a Reference Index Subtable

A reference index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table.

The index subtable row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table.

A maximum of 64 referential constraints can be defined for a table.

Similarly, a maximum of 64 other tables can reference a single table. Therefore, there is a maximum of 128 reference indexes that can be stored in the table header per table.

The limit on reference indexes in the table header includes both references to and from the table and is derived from 64 references to other tables plus 64 references from other tables to the current table = 128 reference index descriptors.

However, the maximum number of reference indexes stored in the reference index subtable for a table is limited to 64, defining only the relationships between the table as a parent with its child tables.

Estimate the size of your reference indexes using the equation provided below.

Reference Index Sizing Equation

RI Subtable Size = (NumDistinct) × (FKLength + PresenceBitsOverhead + VarLengthOverhead + 25)

If fallback is defined for the child table in the relationship, then double the calculated result.

The following parameter definitions are used with this equation.

Parameter	Definition
Row Count * 4	A count of the number of foreign key row references is stored in a reference index subtable. Each foreign key row count is 4 bytes long.
FKLength	The length of a fixed length foreign key value in bytes. Use one of these parameters depending on the reference index in question. <ul style="list-style-type: none"> • If the FK column value is fixed, then use the length of the value. • If the FK column value is variable, then use the average length of the variable length Foreign Key values.
NumDistinct	The value is an estimate of the number of distinct foreign key subtable values. Exclude null foreign keys from this estimate.
Presence Bits Overhead	Use the following equation to calculate this parameter: $\text{Overhead} = \frac{1 + \text{Number nullable FK fields}}{8}$ <p>If there are no presence bits, then the value for this parameter is 0.</p>
VarLength Overhead	Use the following equation to calculate this parameter: Overhead = 2 × (Number variable length FK fields + 1) If there are no variable length foreign keys, then the value for this parameter is 0.
RI Block Overhead	Sum of the following factors. <ul style="list-style-type: none"> • Row length • RI row rowID • Spare byte

Parameter	Definition
	<ul style="list-style-type: none"> • Presence octets • Offsets • Valid flag • Foreign key count • Reference array = 25 bytes

Sizing Spool Space

Vantage uses spool space as temporary storage for result rows that are returned for user requests. Because spool space is a form of temporary space, it is frequently overlooked in capacity planning, yet it is critical to the operations of the database. Spool space needs vary from table to table, user to user, application to application, and with frequency of use. Very large systems use even more spool than smaller systems. Spool rows have a maximum length of approximately 1MB.

Spool falls into these categories:

- Intermediate
- Output
- Persistent
- Volatile

Intermediate Spool Space

Intermediate spool results are retained until no longer needed. You can determine when intermediate spool is flushed by examining the output of an EXPLAIN. The first step performed after intermediate spool has been flushed is designated "Last Use."

Output Spool Space

Output spool results are the final information returned for a query or the rows updated within, inserted into, or deleted from a base table. The length of time output spool is retained depends on the subsystem and various system conditions, as described in this table:

Subsystem/Condition	When Output Spool Is Released
BTEQ	Last spool response.
Embedded SQL	The open cursor is closed.
CLIV2	<ul style="list-style-type: none"> • ERQ received • Function terminated
Session terminates asynchronously due to any number of conditions, including the following. <ul style="list-style-type: none"> • Job abort 	At the time the termination occurs.

Subsystem/Condition	When Output Spool Is Released
<ul style="list-style-type: none"> • Timeout • Logoff 	
System restart	At the time the restart occurs.

Persistent Spool Space

When Redrive protection is enabled, the database stores responses for sessions that participate in Redrive in persistent spool tables. Persistent spools are not deleted following a Teradata restart or node failure. Persistent spools are retained until the SQL request completes and the application has fully received the response. For details about Redrive protection, see *Teradata Vantage™ - Database Administration*, B035-1093.

Volatile Spool Space

The system uses volatile spool space for volatile tables. This is necessary because volatile tables do not have a persistent stored definition.

Sources of Spool Space

Spool space is taken only from disk cylinders that are not being used for data. Data blocks and spool blocks cannot coexist on the same cylinder.

When spool is released, the file system returns the cylinders it was using to the free cylinder list.

Spool Limits

If you find that queries do not run because they run out of spool space, then increase the spool assignment for the user or database having the problem using the MODIFY USER or MODIFY DATABASE statements, respectively. For the syntax and usage notes for these statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. If out of spool space errors are occurring due to uneven data distribution across AMPs, use global space accounting to avoid some of these errors. Global space accounting dynamically allocates and deallocates space to and from AMPs as needed while the underlying tables of the database or user are modified. For more information about global space accounting, see [About Global Space Accounting](#).

The maximum size for a spool row is approximately 1MB. This larger row size enhances DML operations that are limited by small spool rows.

The amount of spool space allocated to each user and database is assigned at CREATE USER or CREATE DATABASE time.

Guidelines for Allocating Spool Space

Unless you reserve a pool of spool space, the space available for spool tends to disappear quickly. When applications consume all the spool space allocated for a system, processing halts.

A large Teradata customer in the retail business uses the following method as a guideline for allocating spool space:

1. Create a special database to act as a spool space reservoir.
Allocate 20% of the total user space in the system for this database.
2. Assign roughly 0.25% of the total space to each user as an upper limit, ensuring that each receives at least as much space as the size of the largest table they access concurrently.

Consider the following factors to perform finer tuning of database, user, or profile spool allotment.

- Query workload types.

Decision support queries generally require more spool than OLTP and tactical queries.

- Average spool use per user.
- Number of concurrent users.
- Number of concurrent queries permitted for any one user.

Spool space is cumulative per user.

- Query size

The smaller the query, the less spool space required. If a particular user only performs small queries, then allocate less spool space to that user.

If a user performs many large queries, then allocate more spool to that user. With the exception of runaway queries, allocating more spool space to a user is never harmful as long as system resources are not wasted.

- Database size

The more AMPs in the configuration, the more thinly spread the data, so the more spool is required per AMP. Because of this, you should consider defining the spool space for a database or user using a constant expression to scale the amount of spool space assigned based on the number of AMPs on the system.

```
CREATE DATABASE spool_reserve AS
  PERM = 2000000*(HASHAMP()+1)
```

where the specified value for PERM space is roughly 20% of the total available disk space for the system, which is based on the multiplier of 2,000,000 bytes.

This specification uses the constant expression $2,000,000 * (\text{HASHAMP}()+1)$ to calculate the number of AMPs in the current system and then scales the PERM space for the *spool_reserve* database to that size.

Sizing a Query Capture Database

Query Capture Databases are used to analyze SQL queries for ways in which they can be better optimized. A QCD can occupy significant storage space, so it is important to understand how much capacity is required to perform query optimization analyses.

Capacity Planning for a Query Capture Database

The following physical limits constrain the QCD functionality:

- Because QCD tables are ordinary Teradata relational tables, they are limited to a row length of 1 MB. Remember that all row lengths must be an even number of bytes (see [Byte Alignment](#)), so be sure to take this into account.
- QCDs draw their disk space from PERM space just like any other persistent Teradata relational tables. Consider defining the spool space for a QCD using a constant expression to scale the amount of spool space assigned based on the number of AMPs on the system.

For example, you might specify the amount of spool space for a QCD as in the following CREATE DATABASE request.

```
CREATE DATABASE qcd AS SPOOL = 2e5 * (HASHAMP() + 1);
```

Note:

After creating the QCD, the spool space does not change if you add more AMPs to your system because the size is based on the number of AMPs in the system at the time the QCD is created, not on the current configuration.

- The row length for tables in an aligned row format QCD is larger in order to align the rows on appropriate modulo(8) boundaries, just as it is for all other base table rows on an aligned row format system (see [Row Structure for Aligned Row Format Systems](#)).

You can estimate the magnitude of the raw data (without indexes) generated by a single INSERT EXPLAIN or DUMP EXPLAIN statement from the following equation:

Raw data size is approximately equal to $(2n + M(t \times d))$

where:

Equation element ...	Specifies the ...
n	size in bytes of the EXPLAIN modifier output for the same query. The approximation assumes that all captured data demographics can be accommodated within $2n$ bytes.
m	size in bytes of the SQL query text.
t	average size in bytes of the DDL text for the object.

Equation element ...	Specifies the ...
d	number of tables and views in the query.

The approximation is slightly more complex when statistics are captured using INSERT EXPLAIN WITH STATISTICS. This equation applies only to INSERT EXPLAIN WITH STATISTICS. DUMP EXPLAIN does not capture statistics.

Raw data size is approximately equal to $(2n + m + (t \times d)) + 5000c$

where:

Equation element ...	Specifies the ...
n	size in bytes of the EXPLAIN request modifier output for the same query. The approximation assumes that all captured data demographics can be accommodated within $2n$ bytes.
m	size in bytes of the SQL query text.
t	average size in bytes of the DDL text for the object.
d	number of tables and views in the query.
c	number of columns involved in range and explicit equality conditions in the query WHERE clause. If you do not specify WITH STATISTICS when you perform INSERT EXPLAIN, the value for c is 0.

The coefficient 5,000 is used as the average size of statistics based on the following information:

- The 101 interval histograms used to store the statistics occupy 4,900 bytes per column. This count includes interval 0 in addition to intervals 1 through 100. Interval 0 contains column- or index-global statistics, while intervals 1 through 200 contain interval-specific column statistics.
- 100 bytes are added to account for an upward bound on miscellaneous bytes per column. Storage overhead in bytes per column due to statistics is approximately equal to $4900 + 100 = 5000$

Related Information

For more information about query capture databases and query optimization analysis, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Sizing Table Space Empirically

The most accurate method for sizing table space for a new configuration is to create a table with a definition as similar as possible to the legacy table you want to port to Vantage. You cannot just extrapolate the space information from your legacy system and expect the result to accurately estimate your Vantage table space needs because commercial database vendors all store their data in different ways.

General Procedure

Use one of the load utilities to load the empty table with a representative sample of rows that constitutes a known percentage of the entire cardinality of the legacy table.

For the first iteration of testing, do not define any secondary indexes on the table. Analyze the space characteristics for this test table, add a secondary index, and repeat the task. Continue this process until you are reasonably certain that you have captured the characteristics of the base table and all the secondary indexes you initially want to define on it.

Extrapolate the total spool space required from an examination of the smaller temporary table. The scaled up data should provide a reliable picture of the initial space requirements for the table and its indexes.

Explicit Procedure

The following table provides a more explicit procedure:

1. Create a new base table on the Teradata platform with a definition that matches the definition for the legacy table it is replacing as closely as possible.
Do not define any secondary indexes on this base table at this stage of the process.
2. Use one of the Teradata load utilities to load the newly defined base table with a representative sample of rows from the legacy table. The loaded sample should not only reflect the demographics of the entire population of the legacy table, but also be a fairly precise percentage of the cardinality for that table.
3. Query the system view (see *Querying DBC.TableSizeV* below) for the space occupied by the new base table.
4. Record the number of bytes returned by the query.
5. Add a secondary index to the table.
6. Query the system view again for the space occupied by the new base table and the new secondary index.
7. Record the number of bytes returned by the query.
8. The arithmetic difference between the numbers you recorded in step 4 and step 7 is the size of the newly defined secondary index.
9. Iterate step 5 and step 6 until you have finished adding all the secondary indexes you anticipate defining for this table.
10. Repeat the procedure with another legacy table until you have estimated the initial size of your entire database.
11. You should also evaluate the sizes of any hash or join indexes (and any secondary indexes defined on join indexes) you anticipate using with the new database. Because hash and join index tables are structurally virtually identical to base tables, you can use the same procedure and the same queries documented by this procedure to determine their size and the size of their indexes.

Querying DBC.TableSizeV

Use the following query to evaluate the sizing of your legacy tables. Assume the database in which the table was created is named `employee_largetable` and the name of the table is `employee`.

```
SELECT SUM(CurrentPerm)
FROM DBC.TableSizeV
WHERE DatabaseName = 'Employee_LargeTable'
AND TableName = 'Employee';
```

The information returned might look something like this:

```
Sum(CurrentPerm)
      2,527,232
```

Table Sizing Summary

The following list reviews the principal points to remember about estimating the table space requirements for your new database.

- Accurate estimates require accurate base data. The minimum base data required for these estimates are the following.
 - Cardinalities (or anticipated cardinalities) of your tables
 - Row size estimates

Remember that all row lengths must be an even number of bytes (see [Byte Alignment](#)), so be sure to take this into account.
- Estimate sizes for all the following database objects.
 - Base tables
 - Base table fallback tables (when defined)
 - LOB and XML subtables
 - Secondary indexes
 - Secondary index fallback tables (when defined)
 - Hash indexes
 - Hash index fallback tables (when defined)
 - Join indexes (including any secondary indexes defined on them)
 - Join index fallback tables (when defined)
 - Spool space
 - Procedures

System-Level Capacity Planning Considerations

This section examines capacity planning for system and table space.

The majority of the material focuses on system space planning, including planning for disk space.

Database sizing issues such as allocating permanent space and estimating database size requirements are also described.

Database Size Considerations

The sizes of the Teradata systems managing relational databases range up to 2,048 CPUs and 2,048 GB of memory supporting 128 TB or larger databases. The BYNET interconnect supports up to 1,084 nodes.

Database sizing considerations must take the following issues into account:

- System disk contents
- Data disk contents
- Data disk space allocation
- Determination of usable data space

[Teradata System Limits](#) contains information about the various minimum and maximum sizes of database objects.

Teradata Secure Zones inside a Database

Database designers can create one or more exclusive database hierarchies, called zones, within a single Teradata system. Access to the data in each zone and the zone administration is handled separately from the Teradata system and from other zones.

Teradata Secure Zones are useful in situations where the access to data must be tightly controlled and restricted. You can also use Teradata Secure Zones to support regulatory compliance requirements for the separation of data access from database administration duties.

For example, consider the following use of Teradata Secure Zones. Suppose you have a multinational company or conglomerate enterprise with many subsidiaries. You can create a separate zone for each of the subsidiaries. If your company has divisions in different countries, you can create separate zones for each country to restrict data access to the personnel that are citizens of that country. Your corporate personnel can manage and access data across multiple zones while the subsidiary personnel in each zone have no access to data or objects in the other zones. A system-level zone administrator can manage the subsidiary zones and object administration can be done by either corporate DBAs or zone DBAs, as required.

With Teradata Secure Zones, you can make sure of the following:

- Users in one subsidiary have no access or visibility to objects in other subsidiaries.
- Corporate-level users may have access to objects in any or all subsidiaries.

Another typical scenario is the case of cloud companies that host multiple data customers as tenants. Companies that offer cloud-based database services can host multiple tenants within a single Teradata system, using zones to isolate the tenants from each other as if they were running on physically segregated systems. Zone DBAs can administer the objects in their own zone as required. Teradata Data Warehouse may manage tenant zones if the shared system is Teradata-owned.

The tenant zones can be managed by a system-level zone administrator, where Teradata acts as the system administrator.

With Teradata Secure Zones, you can make sure of the following:

- Users in a tenant zone have no access or visibility to objects within other zones.
- Users in a tenant zone cannot grant rights on any objects in the zone to any other users, databases, or roles of other zones within the system.

Zones are stored in the DBC.Dbase table. Therefore, they fall under the system limit for the maximum number of combined databases, users, and zones (see [System Limits](#)).

For more information, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

System Disk Contents

System disks Disk 0 and Disk 1, are cabled to a controller not associated with the disk arrays.

Disk 0 is the boot disk. The slices on Disk 0 contain file systems under the control of the operating system root directory.

Disk 1 provides additional space for dumps and memory swapping.

Boot Disk Contents

The following table lists the contents of the boot Disk 0.

Content	Use
AMP identifiers file	Stores identifiers of all AMPs.
Configuration maps	Defines vprocs. The configuration map lists the pdisks allocated to a clique, but does not assign them to vdisks or AMPs.
Open PDE and operating system	Stores software under which Vantage is running.
Executable Teradata software	Includes any optional Teradata client software packages.
UDF libraries	Support user-defined functions and external stored procedures.
Copy of Teradata GDOs	Includes DBS Control Record (DBSCONTROLGDO).
Values	Initialize hash buckets.

Content	Use
space	<ul style="list-style-type: none"> • Diagnostic reports • Memory swap space • Operating system and Open PDE dumps

Data Disk Contents

The virtual data disks controlled by the AMPs include reserved space and permanent space.

Reserved Space

Reserved space is used for the TVS map, DeviceTag, and statistics areas that are located at the beginning of each pdisk.

Permanent Space

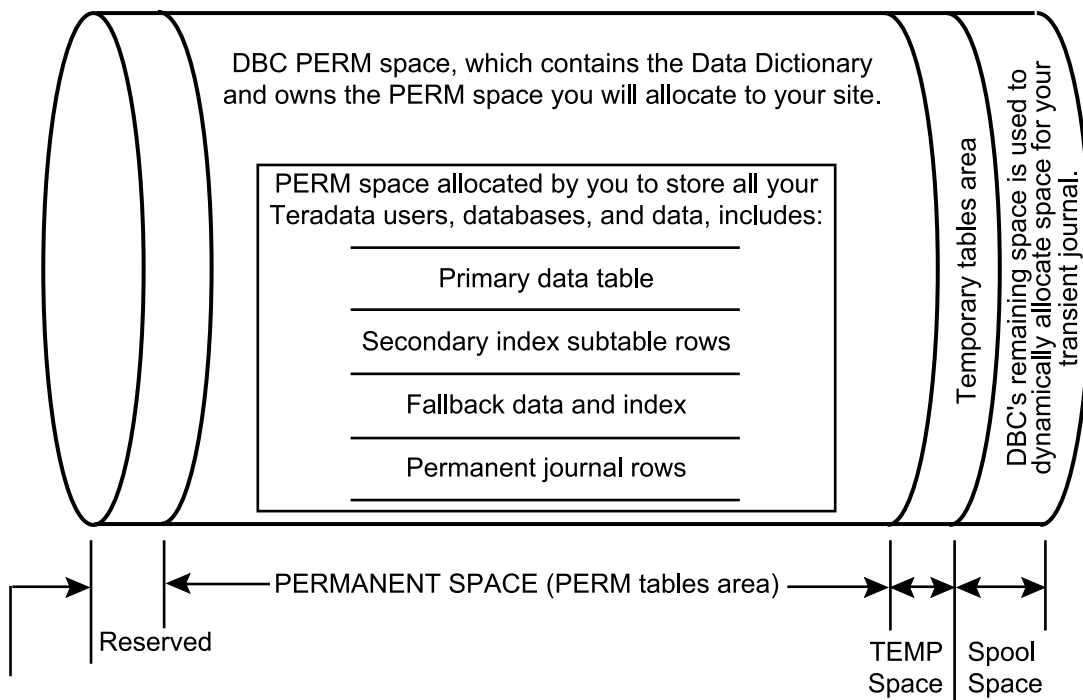
Content	Use
CRASHDUMPS user	Temporary storage of diagnostic information and crash dumps generated by the database and Open PDE.
SYSTEMFE user	Special macros and objects used by Teradata field support personnel.
SYSADMIN user	Special views and macros used by Teradata Services personnel.
TDStats	Contains all of the tables, views, macros, UDFs, and external stored procedures used by the Teradata statistics management system. TDStats is the repository for statistics management data.
Data dictionary	Maintains Vantage user, database, table, index, trigger, permanent journal, view, stored procedure, and macro definitions.
WAL log (transient journal) space	Stores Redo and Undo records that enable data to be brought to a consistent state as well as before-image records that enable uncommitted transactions to be rolled back.
Depot area space	Stores cylinders reserved by the File System as a staging area to temporarily contain modified-in-place data blocks before they are written to their home disk destinations.
Spool	Stores intermediate and response results of SQL requests.
Temporary space	Stores data inserted into materialized global temporary tables. Note that each materialized global temporary table also requires a minimum of 512 bytes from the PERM space of its containing database or user for its table header.
Top-level hierarchical ownership of PERM and TEMP spaces	<ul style="list-style-type: none"> • Databases • Base data tables • Join index subtables

Content	Use
	<ul style="list-style-type: none"> • Hash index subtables • Fallback data and join index subtables • Secondary index subtables • Permanent journal tables • Stored procedures • Table headers

Data Disk Space

Data disk space is reserved for system use, PERM space, or TEMP space owned by user DBC, as illustrated in the following graphic. For purposes of this graphic, consider join index and hash index subtables to be identical to primary data tables.

The size considerations for each type of space are described in [Permanent Space Allocations](#).



Permanent Space Allocations

Permanent data disk space includes database and user fixed PERM allocation. The database initialization routine creates these system users:

- DBC (at the top of the hierarchy)
- SYSTEMFE (owned by DBC)

- SYSADMIN (owned by DBC)

All permanent space not consumed by SYSTEMFE and SYSADMIN is allocated to user DBC. The ramifications of the space allocation for each of the three system users are discussed in the subsections that follow.

System User DBC

The PERM space allocation of system user DBC is used for the following:

This DBC space...	Performs this service ...	For this function ...
PERM	provides ownership	<p>Teradata production database space, which must accommodate the maximum PERM space allocated to each user and database by its creator.</p> <p>The allocated space is used to store the rows in the data tables created by each user, including the following items.</p> <ul style="list-style-type: none"> • Primary data table rows • Join index table rows • Hash index table rows • LOB and XML subtable rows • Secondary index subtable rows • Optional permanent journal image rows • Optional fallback rows, which replicate the rows of primary data tables, index subtables, hash and join index tables, and permanent journals
	creates and maintains the WAL log/transient journal	storing a before-image copy of every row affected by data modifications in all current sessions and Redo and Undo WAL records.
	stores system table, index, macro, trigger, and view definitions	setting up the data dictionary.
	provides permanent space	<ul style="list-style-type: none"> • Table headers. Note that each materialized global temporary table also requires a minimum of 512 bytes from the PERM space of its containing database or user for its table header. • CRASHDUMPS database.
TEMP	provides temporary space	data inserted into materialized global temporary tables.
SPOOL	provides maximum defined SPOOL space per creator-defined allocation to each user and database	<ul style="list-style-type: none"> • Dynamic allocation and release of intermediate and final result sets during query processing. • Volatile temporary tables.

System User SYSTEMFE

System user SYSTEMFE is owned by user DBC. SystemFE contains special macros and objects. The quantity of PERM space allocated to SYSTEMFE is minimal.

Only Teradata field support personnel normally log on under this user name.

System User SYSADMIN

System user SYSADMIN is owned by DBC. SysAdmin contains special views and macros. The quantity of PERM space allocated to SYSADMIN is minimal.

Only Teradata field support personnel normally log on under this user name.

Estimating Database Size Requirements

Allocating Space for External Stored Procedure and User-Defined Function Bodies

The code for both stored procedures and user-defined functions is stored outside the database on system disk, or, in the case of UDFs, possibly on client disk, so you do not need to account for it when you undertake database capacity planning.

Allocating PERM, TEMP, and Spool Space

Use the CREATE USER statement to allocate all the PERM space required by the applications that support your enterprise to user SysAdmin. To determine these space requirements, use the equations provided in [Calculating Total PERM Space Requirements](#).

Note:

If the Data Dictionary and the Crashdumps user have not yet been created, run the DIP utility and then check the remaining PERM allocation before completing space calculations.

The quantity of space specified is allocated from the current PERM space of user DBC. DBC becomes the owner of the SysAdmin user and of all users and databases subsequently created. Be sure to leave enough space in DBC to accommodate the growth of system tables and logs and the WAL log.

Some guidelines for estimating temporary spool and WAL log space requirements are explained in the following subsections.

Note that the containing database or user of a global temporary table must also have a minimum of 512 bytes of PERM space free for the GTT table header.

Estimating Administrative Spool Space Requirements

As with PERM space, spool space is allocated from the available space of the owning user. The SPOOL specification in the CREATE request is only a maximum limit.

During query processing, spool space is dynamically allocated from any free space, including the free PERM space of the user who submitted the query.

In general, the following guidelines apply.

- Reserve 25% to 35% of total space for spool space and spool growth buffer.
When you create user SysAdmin, you can leave the SPOOL parameter unspecified, so it defaults to the maximum allocation of the owning user, user DBC.
- Allow an extra 5% of PERM space in user DBC.
- Each time a new user or database is created, you can specify the maximum amount of spool space that a query submitted by that user can consume.

Details and formulas for estimating user spool space requirements are provided in [Tables Area: User Spool Space Requirements](#), [Rules for Using the Spool Space Equations](#), [Field Mode Spool Space Sizing Equation](#), and [Record and Indicator Mode Spool Space Sizing Equation](#).

Estimating Administrative TEMP Space Requirements

Allocate the TEMP space required by the applications that support your enterprise to user SysAdmin.

Like PERM space, TEMP space is allocated from the available space of the owning user. Make an estimate of temporary space requirements.

The TEMPORARY=n clause in the CREATE DATABASE and CREATE USER statements permits you to define how many bytes are to be allocated to a database or user for creating global temporary tables. Note that temporary space is reserved prior to spool space for any user defined to have this attribute.

Disk usage for materialized global temporary tables is charged to the temporary space allocation of the user who referenced the table.

Global temporary tables also require a minimum of 512 bytes from the PERM space of the containing database or user. This space is used for the GTT table header.

If no temporary space is defined for a user, then the space allocated for any global temporary tables referenced by that user is set to the maximum temporary space allocated for the immediate owner.

Estimating WAL Log Space Requirements

Vantage creates and manages a WAL log that includes the transient journal to store the before-change image of every data row involved in every SQL transaction. This log is used to recover data tables when transactions are aborted.

As transactions are processed, the WAL log grows and shrinks. While transactions are in progress, the WAL log grows according to the total number of data rows being updated or deleted.

The applicable journal rows are purged at intervals (but not before the transaction is completed or, if it was aborted, not before the affected rows are recovered).

Space for the WAL log is acquired from the current PERM space of user DBC. Therefore, it is important that you leave enough PERM space in DBC to accommodate the growth of the largest foreseeable WAL log.

To estimate the maximum size of the WAL log, follow this procedure:

1. Determine the length of the longest row in your production database. Use this figure as your maximum row length.
2. Multiply the maximum row length by the total number of rows in your application programs, batch jobs, and ad-hoc queries that users are likely to update and delete in concurrent transactions.
3. Double the resulting value.

This is the space needed for the WAL log on your system.

WAL Log Data Block Size

When you know the maximum row length, double the value to calculate the size of your multirow data block definition in the JournalDBSize parameter of the DBS Control Record. For example, the JournalDBSize value should be greater or equal to twice the maximum row length.

The file system allocates any row that exceeds the current size of a multirow data block to a WAL log data block of its own.

Depending on the configuration of your system, the database builds its data blocks using either native 512 byte sectors or native 4KB sectors. Systems built on 512 byte sectors are referred to as unaligned configurations, while systems built on 4KB sectors are referred to as aligned configurations.

For details on the JournalDBSize parameter, see the DBS Control utility section in *Teradata Vantage™ - Database Utilities*, B035-1102.

About Global Space Accounting

DBAs can adopt one of two strategies for managing permanent, spool, and temporary space:

- AMP level only. The per-AMP space quota is the maximum permissible AMP-level space. This is the default.
- Global level, which maintains both an overall system limit and AMP limits. This strategy can provide extra space to AMPs when needed by temporarily reducing space for AMPs that do not need the space now.

When DBAs manage space at the AMP level, transactions or long-running load jobs abort when space use exceeds hard limits set for the database or user. If a DBA manages space at the global level, two allowances can be made to increase space to AMPs when needed:

Space Allowance	Creation Method	Description
Skew factor	<p>The SKEW option in a CREATE USER/DATABASE or MODIFY USER/DATABASE request.</p> <p>If the SKEW option is not specified or is specified as DEFAULT, the skew factor is controlled by three DBS Control utility fields:</p> <ul style="list-style-type: none"> • DefaultPermSkewLimitPercent: Controls the skew factor on permanent space • DefaultSpoolSkewLimitPercent: Controls the skew factor on spool space 	<p>Gives extra space to one or more AMPs when needed, up to the value of the SKEW option. The skew option can be defined as a percentage or a constant expression. You can apply this option to PERM, SPOOL, and TEMP space.</p> <p>A skew limit value of 0 means that the AMP space quota is the space limit, and Vantage does not provide more space to AMPs, even when other AMPs have space available. A</p>

Space Allowance	Creation Method	Description
	<ul style="list-style-type: none"> DefaultTempSkewLimitPercent: Controls the skew factor on temporary space 	<p>non-zero soft limit percent permits this quota to increase by the soft limit percent. Users receive an error if actual space use exceeds the soft limit. The default value for the default skew limit percent fields is 0.</p> <p>The DBC.GlobalDBSpace table records the cumulative allocations of the AMPs and maintains database and user global-level space limits.</p>
Global soft limit	Set a non-zero value for the GlobalSpaceSoftLimitPercent field in the DBS Control utility	<p>The percentage by which a database or user is allowed to exceed the maximum space limit. This setting applies to PERM, SPOOL, or TEMP space. The default value is 0.</p> <p>The system sends alerts when processing exceeds soft limits, so the DBA can take corrective action. Processes that cannot be aborted (such as transaction recovery) may be in progress when the soft limit is exceeded or when physical storage limits are almost reached. In this case, the system continues processing but reports alerts with higher severity levels. The DBA is expected to increase affected space as soon as possible in this situation.</p>

Note:

Teradata recommends that you change the values of the default skew limit percent fields and the GlobalSpaceSoftLimitPercent field only under the direction of Teradata Support Center personnel.

Create a User with a Skew Limit

The following example creates a user with a 10 percent skew limit for permanent space and a 20 percent skew limit for spool space:

```
CREATE USER Caspian AS PERM = 1e9 SKEW = 10 PERCENT, SPOOL = 2e9 SKEW = 20 PERCENT;
```

In the preceding example, if the system has 4 AMPs, each AMP has a limit of 250 MB of permanent space. With the permanent skew limit of 10 percent, any AMP can exceed the permanent space limit by 25 MB. An AMP may use up to 275 (250 + 25) MB of permanent space, as long as the total permanent space used by all AMPs does not go beyond the 1 GB global limit.

Similarly, each AMP has a limit of 500 MB of spool space. With the spool skew limit of 20 percent, any AMP can exceed the spool space limit by 100 MB. An AMP may use up to 600 (500 + 100) MB of spool space, as long as the total spool space used across all AMPs does not exceed the 2 GB global limit.

Considerations for Global Space Accounting

Consider the following factors about global space accounting:

- Teradata recommends global space accounting for a database or user when actual space use is expected to be non-uniform, for example, when large, unknown data sets that may be skewed are often loaded into the underlying tables or when the database or user has stored procedures or UDFs that record very few rows with object-specific data.
- AMP-level space accounting does not have the overhead of dynamic, need-based space allocation and is recommended when the database is uniformly distributed across all AMPs in the system or when the global limit is high enough to allow for variability in data distribution due to skew .
- An unlimited skew limit for spool space may not be a good option, since design issues or a lack of statistics may cause a request to use excessive spool on some AMPs. A lower spool limit causes requests using excessive spool space to abort.
- To reduce the chance of space shortages causing aborted transactions, consider setting a higher value for the global soft limit if the system has long-running transactions that consume a lot of space.
- Skew limits cannot solve space problems, especially when actual use is close to the limits. If there is not enough space, it is best to increase the space limits instead of allowing more skew.

Interaction between Skew Factor and a Profile

- Profile values supersede user values. For example, if a spool limit is defined in both the CREATE USER request and the profile a user is a member of, Vantage uses the profile value.
- If a user is a member of a profile, the skew factor is controlled by the user definition.

For example, if a CREATE USER request gives user Joe 10 MB spool with 20% skew, but the profile Joe belongs to provides 5 MB spool, then Vantage applies the 5 MB spool limit with 20% skew. The same is true for temporary space specifications.

Interaction between the Skew Factor and Soft Limits

Skew factor and soft limits can be cumulative. For example, consider the following numbers:

- Global limit: 800 MB
- Soft limit: 10 percent
- Per-AMP limit on a 4-AMP system: 200 MB
- Skew factor: 25 percent

With these numbers, the global limit can exceed the 800 MB limit by 10 percent (to 880 MB). In addition, an AMP can exceed the 200 MB limit by 25% (to 250 MB) if the total across all AMPs does not exceed 880 MB. Each of the AMPs can also receive an additional 20 MB because of the global soft limit. The cumulative effects of the skew factor and global soft limit can increase the maximum possible space allotment to an AMP to $250 + 20 = 270$ MB, as long as the total space used across all AMPs does not exceed 880. The user will receive an error message if either of the following limits are exceeded:

- The global limit of 880
- The per-AMP limit of 270

Related Information

For more information on...	See...
the SKEW option for DDL in CREATE DATABASE and CREATE USER or MODIFY DATABASE and MODIFY USER	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
DBS Control fields	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.

Determining Available User Table Data Space

To determine available user table data space, you must take several non-user table data space allotments into account.

First you must determine how much space these allotments account for and then you must subtract them from the total available table space.

The nonuser table space allotments that must be accounted for are those in the following list.

- Overhead space, including space reserved for allocation maps and related statistics.
- Depot area, which includes cylinders reserved as a staging area for modified-in-place data blocks before they are written to their home disk destinations.

Note:

Data blocks that are not modified in place are not written to the depot area.

- Tables area, which includes the following.
 - Data Dictionary (for the WAL log and system tables)
 - Crashdumps user
 - User temporary space
 - User spool space

The following topics describe specific space requirements and provide equations with which to determine variable space requirements.

Depot Area Overhead

The Depot consists of two types of slots:

- Large Depot slots
- Small Depot slots

The Large Depot slots are used by aging routines to write multiple blocks to the Depot area with a single I/O.

The Small Depot slots are used when individual blocks that require Depot protection are written to the Depot area by foreground tasks.

The number of cylinders allocated to the Depot area is fixed at startup. Consult your Teradata support representative if you want to change this value.

The number of Depot area cylinders allocated is per pdisk, so their total number depends on the number of pdisks in your system. Sets of pdisks belong to a subpool, and the system assigns individual AMPs to those subpools.

Because it does not assign pdisks to AMPs, the system calculates the average number of pdisks per AMP in the entire subpool from the vconfig GDO when it allocates Depot cylinders, rounding up the calculated value if necessary. The result is then multiplied by the specified values to obtain the total number of depot cylinders for each AMP. Using this method, each AMP is assigned the same number of Depot cylinders.

The concept is to disperse the Depot cylinders fairly evenly across the system. This prevents one pdisk from becoming overwhelmed by all the Depot writes for your system.

Data Table Overhead

The system uses some amount of data space, which includes permanent, spool, and temporary space, as overhead for various purposes.

The percentage of data space required for cylinder indexes is calculated based on the following information:

- Each cylinder has 2 cylinder indexes.
- Each cylinder index is 32 KB (64 sectors) in size.

Therefore, 128 sectors are reserved for cylinder indexes per cylinder.

- Each cylinder is 23,232 sectors in size.

The quantity of permanent data table space, temporary space, and spool space available on a system is also limited by the amount of disk space required by WAL to process update transactions. The amount of disk required by WAL is a function of the number of update operations being undertaken by the workloads running on the system at any given time, and waxes and wanes as a result. The more updates that are done at a time, the greater the quantity of WAL data that is produced. The amount of WAL data produced is roughly equivalent to twice the amount of transient journal information that is required for updates.

Allow space for data table overhead as listed in the following table:

Number of Cylinders	Purpose
13	<p>General</p> <p>One cylinder each for the following purposes:</p> <ul style="list-style-type: none"> • Permanent space sentinel • Permanent journal space sentinel <p>System journals are stored either in user DBC, which is permanent space, or in the WAL log, which is not part of journal space.</p> <ul style="list-style-type: none"> • Global temporary spool space sentinel • Spool space <p>Nine cylinders for the following purpose:</p> <ul style="list-style-type: none"> • Write Ahead Logging <p>The additional cylinders for WAL are required to ensure that MiniCylPack is able to run in low disk situations to free space.</p>

Number of Cylinders	Purpose
Approximately 1.25% of the available data space (minimum)	Cylinder indexes
A minimum of 7% free space per cylinder	Fragmentation You must be careful not to allow too much free space for fragmentation, but you must be equally careful not to allow too little. The most practical way to establish an optimal number of cylinders to reserve for fragmentation on your system is trial and error.
Ranges between 1 and 20 per pdisk.	Depot The number of cylinders assigned to Depot is set at startup. The value for your system can only be changed by Teradata Services personnel.

Tables Area: Dictionary and Spool Space Requirements

After you determine how much space remains when overhead is accounted for, determine how much space is required for the data dictionary and spool.

You should reserve approximately 80 MB for growth of system tables and the WAL log. However, optimum space for the WAL log should be based on how many rows are dropped or updated by concurrently running transactions during your peak workload.

Tables Area: User TEMP Space Requirements

The TEMPORARY clause in the CREATE DATABASE and CREATE USER statements permits you to allocate default space within this database or user for inserting data into global temporary tables materialized by users. Note that Vantage always reserves temporary space prior to spool space for any user defined with this attribute.

Disk usage for a materialized global temporary table is charged against the temporary space allocation of the user who referenced the table.

Global temporary tables also require a minimum of 512 bytes from the PERM space of the containing database or user. This space is used for the GTT table header.

If no default temporary space is defined for a database, then the space allocated for any global temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.

Subtract the TEMP amount allocated to this database.

Tables Area: CRASHDUMPS User Space Requirements

Subtract the PERM amount allocated to this user. The DIP utility creates this database with a default allocation of 1 GB.

If you modified the default amount when you set up the Crashdumps user, subtract the actual current amount.

Tables Area: User Spool Space Requirements

Whenever you create a new user, reserve a minimum of 20% of user permanent space for spool space. It is better to determine the optimum amount of space according to your application environment. This takes into account such factors as average table size, protection choices, number of rows involved in commonly used transactions, and so forth.

The optimum allocation depends on your application environment. If your requirement is to perform the calculations according to the formulas provided in this section, and the calculated amount exceeds 20% of permanent space, use the actual figure plus 5%. To determine an allowance based on your applications, use the formulas in this section.

Rules for Using the Spool Space Equations

When using the spool space equations, keep the following rules in mind:

- The equations are based on the following variables.
 - Number of rows in the spool
 - Amount of data in each row
 - Mode of select operation (Field or Record)
- The equations only determine the space needed to return rows to the user.
- Response rows are limited to approximately 64,000 bytes.
If the row is longer, a row length error is reported.
- Even if an ORDER BY clause is not included, the system creates a minimum length sort key of 8 bytes for both Record mode and Field mode select operations.
- All descriptions of CHARACTER columns also apply to BYTE columns.
- Be sure the NF, CF, SCF, and RDS values represent the sums of all selected or sorted columns.
For example, a FORMAT phrase can cause blanks to be added, which must be figured into the total column length.

Field Mode Spool Space Sizing Equation

Use the following equation to determine the amount of usable data space for Field mode:

$$\text{Usable Data Space} = a(\text{RO} + \text{RP} + n(\text{PH} + \text{NF} + \text{CF})) + b(\text{SNF} + \text{SCF})$$

The following parameter definitions are used with this equation:

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (22 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).
n	Number of columns being selected.

Parameter	Definition
PH	Parcel headers (4 bytes per field).
NF	Formatted size of each numeric column (spaces, dollar signs, and commas should be included).
CF	Formatted size of each character column. For example, if a selected column is defined as VARCHAR(200), it takes up 200 characters even if it contains only 3 nonblank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).
SCF	Formatted size of each character column in the ORDER BY clause. If an order operation is done on a column defined as VARCHAR(200), 200 characters are allowed, even if there are only 3 nonblank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.

Record and Indicator Mode Spool Space Sizing Equation

Use the following equation to determine the amount of usable data space for Record and Indicator modes:

$$\text{Usable Data Space} = a(\text{RO} + \text{RP} + \text{RDS} + (\text{b}(\text{SNF} + \text{SCF})) + \frac{n}{(\text{IF} + 1)})$$

The following parameter definitions are used with this equation:

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (12 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).
RDS	Raw data size of each field returned to the user.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).
SCF	Formatted size of each character column in the ORDER BY clause. If an order operation is done on a column defined as VARCHAR(200), 200 characters are allowed, even if the value contains only 3 nonblank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.

Parameter	Definition
n	Number of columns being selected.
IF	Record mode selects use indicator variables to represent nulls. IF is the constant 8, meaning that for each 8 columns selected, 1 byte is needed to represent a null column.

Calculating Total PERM Space Requirements

Use the following procedure to determine how much PERM space to allocate to user SysAdmin.

1. Estimate the size of each database, as follows.
 - a. Estimate the size of the primary data table, including fallback (see [Sizing Base Tables, LOB Subtables, XML Subtables, and Index Subtables](#) for instructions for how to do this) and LOB and XML subtables.

LOB and XML values are also stored in the permanent journal, so LOB and XML column space must account for additional permanent journal append and storage costs in addition to LOB and XML subtable storage costs.
 - b. Estimate the size of unique and nonunique secondary index subtables, join indexes, and hash indexes.
 - c. Add the primary data table estimate and index and LOB and XML subtable estimates (including fallback) to obtain the estimated total table size.
2. Estimate the total table storage by adding together the space estimates of all table sizes (Substeps a, b, and c). Use this sum for step 3.
3. Estimate the space requirement for the application.

Note:

If the calculated sum is greater than the remainder calculated in step 4, contact your Teradata sales representative to discuss a system expansion.

- a. Add 7% for fragmentation.
- b. Add extra space for spooling, TEMP space, and for variability in hashing. Although this space requirement varies according to the applications, adding 20% to 30% is usually adequate if you do not use permanent journaling.
- c. Estimate the size of each journal table per database.
- d. Note that if journaling is used, the extra space requirement depends on two factors:
 - Size of the data tables that write to that journal
 - Number of changes applied to those tables before the journal table is dropped
- e. Add the sums calculated for steps 1 through 4 and use their sum for the calculation in step 5.
- f. For each disk, add 3 cylinders per disk as spaces and allow 2% of the formatted space for cylinder indexes.

4. Subtract the amount of space needed to accommodate user DBC contents plus your maximum WAL log from the user DBC PERM space.

Specify about 80 MB for the growth of system tables and the WAL log plus another 5% of the current PERM space for spool growth.

Use the remainder for step 5.

5. If possible, subtract the sum of step 3 from the result of step 4.
6. Deduct the number of cylinders that make up the default free space specified by the DBS Control record for operations that use the percent freespace (FSP) value.

For example, if your FSP is set to 15, deduct an additional 15%.

If you plan to create large tables with a freespace percentage greater than the GDO default, then use the larger percentage.

For example, if you define your largest tables with an FSP of 30, then deduct an additional 30%.

The remainder is the amount of PERM space you should allocate to the SysAdmin user.

Note:

Table sizing is normally the responsibility of Teradata field support personnel.

Designing for Backups

There was a time not so very long ago when backups were a minor issue in database management. You set up a batch job to perform the backup overnight, while the system was not being used by anybody else, and that was that. Several issues have complicated that once common scenario.

First is the tremendous increase in the size of databases that can be supported by a relational database management system like Teradata. Where at one time a “very large database” might be on the order of several GB, it is increasingly common to see multiterabyte databases supporting large data warehouses and support for petabyte and even yottabyte databases is on the near horizon. Magnetic tape is a serial medium. Even if you archive to multiple tape drives simultaneously, there is still a considerable time issue involved with backing up multiterabyte databases.

Second is the end of the era of the batch window. Enterprise data warehouses now typically support worldwide operations, and that means that the system must be available to users 24 hours per day, seven days a week. Without the old batch window, how do you back up your business-critical data without having a significant negative effect on the capability of your data warehouse to support its worldwide user base?

Many approaches to solving these problems fall outside the scope of database design, but at least one method for minimizing the impact of backing up very large databases can be designed into your database. That method is the subject of this topic.

Terminology

For purposes of this topic, large and small tables and databases are defined as follows:

Term	Definition
Small table Small database	Less than 100 GB.
Large table Large database	Greater than 100 GB.

Product Issues

The issue described in this topic and its suggested solution apply to the large system environment, primarily when using the NetVault product to backup to and recovery from different machines. The REEL product makes it possible to restore to or recover from the same machine using its fast path option, but even that option does not relieve the problem of backing up to a second machine.

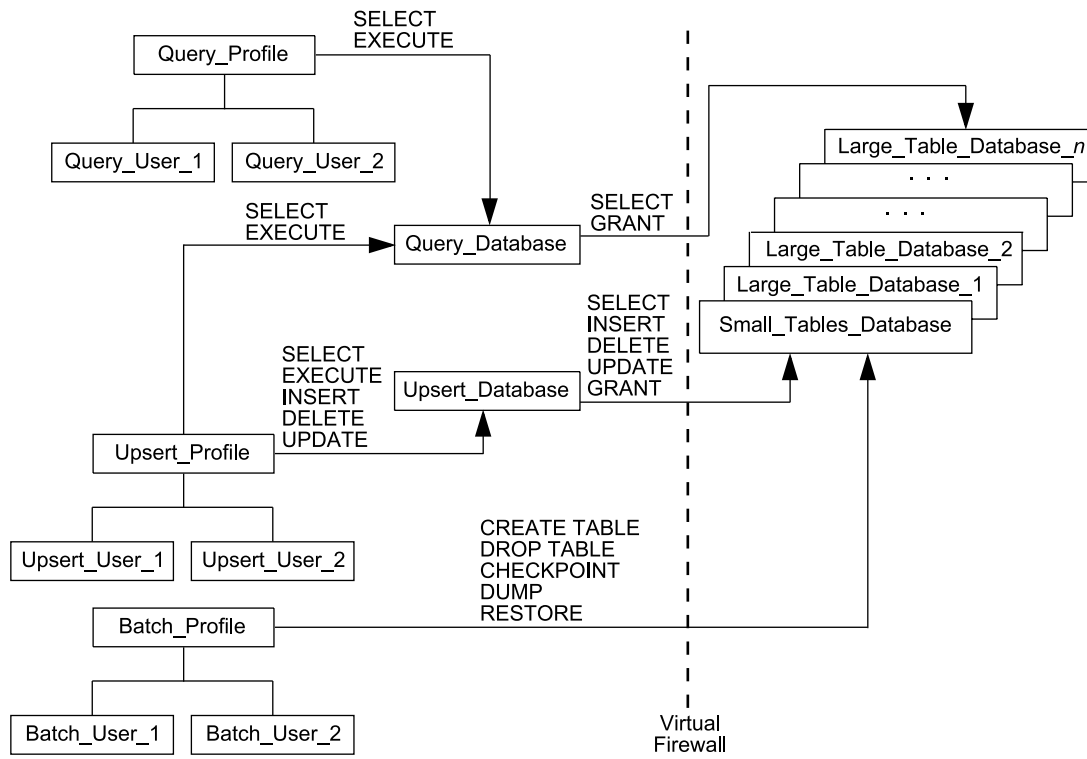
The Scenario

Database and table dumps and restores are a time consuming process. To minimize this problem, you can isolate large tables in their own databases in order to isolate their backup and restoration. Otherwise, when tables of this magnitude are mixed with much smaller tables, you have two choices:

- Back up the large table in one pass and then backup the remaining tables in a second pass.
- Back up the entire database at once, which takes a very long time to complete (and even longer to restore)

A Solution

Design your database in such a way that large tables are isolated within their own, separate databases, then build views for your users to use to access the data. Carefully constructed views can mask this separation and permit users to access data across multiple databases transparently within a single query. The following graphic illustrates one possible design that uses this method.



Design Issues for Tactical Queries

The majority of the topics covered by this document apply to designing for strategic, or decision support, queries. Because you can also run your operational data store on Vantage, this section highlights some of the design issues you must consider when implementing a physical design to support mixed workloads of both strategic and tactical queries.

The document discusses other material related to more specific design issues for tactical queries in [Primary Index, Primary AMP Index, and NoPI Objects](#) and [Join and Hash Indexes](#).

The section does not make suggestions about scheduling job priorities to achieve an optimal mix of strategic and tactical workloads for your system, nor does it describe the various locking issues associated with tactical queries.

See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for information about lock management for tactical queries.

Tactical Queries Defined

Traditionally, data warehouse applications have been based on drawing strategic advantage from the data. Strategic queries are often complex, sometimes long-running, usually broad in scope. The Vantage parallel architecture supports these types of queries by spreading the work across all of the parallel units and nodes in the configuration.

As data warehouse users require more versatility out of their data, they are supplementing their strategic focus with a tactical component. Today, increasing numbers of Vantage users are introducing their quick turnaround tactical queries alongside the classic strategic queries in their Teradata system.

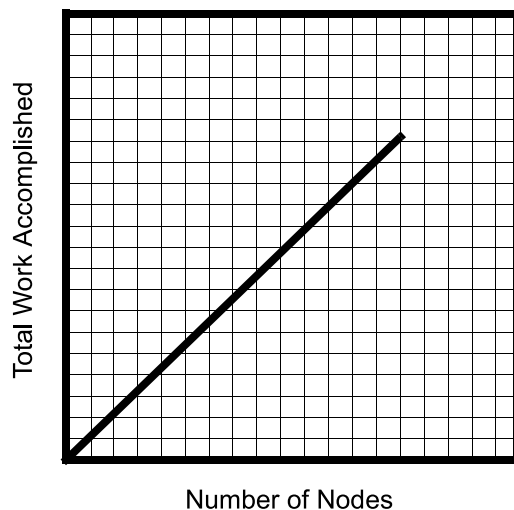
The name tactical query is given to a certain category of work that is short, highly tuned, and focused on more operational decision-making. This class of queries is sometimes categorized under the initialism OLCP, which stands for Online Complex Processing. Tactical queries are similar to classical online transaction processing (OLTP), in that both are composed of short, often direct-access queries, coming with defined response time requirements.

OLTP, however, is more focused on bookkeeping activities and recording operational events, while tactical queries are more focused on decision-making, recording selections or choices made. OLTP generally comes with a higher ratio of writes compared to reads, while tactical queries tend to be more read intensive. OLTP usually has a departmental view and a very narrow context, while tactical queries are often more enterprise-wide. With OLTP queries, short latency is always critical. Tactical queries, on the other hand, are designed for response time requirements that can vary from subsecond up to 10 or 20 seconds or more.

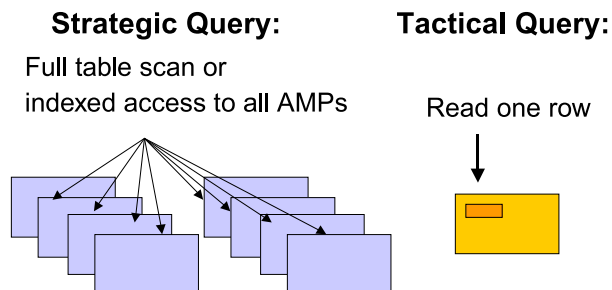
Scalability Considerations for Tactical Queries

Scalability Is a Relative Concept

In traditional decision support applications, setting the stage for scalable performance with Vantage involves distributing the work equally across all the AMPs in the system. As more nodes are added to a system, the number of AMPs increases proportionally, and the effort to process a complex query is spread across more parallel units, reducing response times. The following plot indicates linear scalability, graphing the total workload accomplished as a function of the number of nodes in the system. Strategic queries run proportionally faster as more nodes are added to a system.



Spreading the data evenly across all AMPs remains the goal of designing the database to support tactical queries. But to achieve the type of scalability that best supports tactical queries, you need to localize the work in such a way that it accesses only a few rows using the least resources possible. This means accessing the smallest number of AMPs possible to perform an operation; optimally only one. The following figure indicates that when tactical queries access only a few AMPs, more of them can run concurrently.



Effect of Data Volume Growth on Tactical Query Response Times

Query times for complex decision support are directly impacted by increases in the data volume. Because a large portion, if not all, of a table might need to be read, if that table doubles in size, then the effort required to process the work can also double. On the other hand, a query that accesses one or a few rows in the database is not impacted by changes to the data volume. In most cases, a tactical query accesses one or few rows whether the table those rows are in is 1 GB or 10 TB in size.

Effect of Growth in Concurrent Users on Tactical Query Response Times

Raising the level of concurrent users doing the same work in traditional decision support tends to slow response times for the current users because more demands are now being made on the same resources, and all are spread out across all nodes and AMPs in the system. In contrast, increasing the number of users performing tactical queries that are very localized and limited in their resource use boosts overall throughput up to the point of system saturation.

Effect of Configuration Expansion on Tactical Query Response Times

For a complex query, adding nodes translates to a proportional decrease in response time because the work is distributed across a larger number of AMPs. At the same time, the response time for a query that accesses a single AMP, as many tactical queries do, is not affected by the number of AMPs in the configuration.

There is a one benefit gained by single-AMP tactical queries when nodes are added to a configuration: more of them can be performed at the same time and still deliver short turnaround times.

Consider the following contrived example: Assume you have a table with a cardinality of 1 million rows. The read capacity of each node in your configuration is 100 rows per second. If you are doing a complex strategic query that involves a full table scan of this table, then the response time for the query diminishes proportionally with the increase in nodes, as illustrated by the data in the following table:

Number of Nodes	Number of Rows per Node	Response Time (seconds)
1	1,000,000	10,000
10	100,000	1,000
100	10,000	100
200	5,000	50

This performance enhancement occurs because each node has fewer rows as more nodes are added, so each node can perform its portion of the scan faster.

On the other hand, if your application is performing single- or few-AMP tactical queries, adding nodes does not shorten the response time. However, it does increase the number of tactical queries that can be performed in a given interval of time, as indicated by the data in the following table:

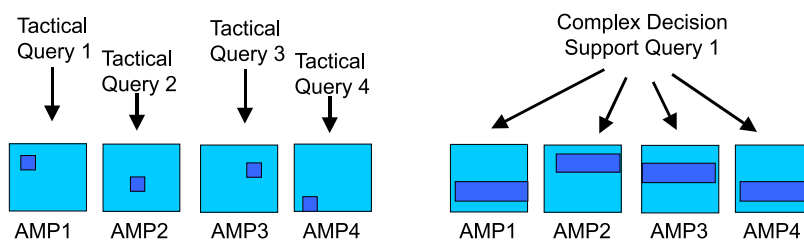
Number of Nodes	Response Time (seconds)	Throughput (requests/second)
1	0.01	100

Number of Nodes	Response Time (seconds)	Throughput (requests/second)
10		1,000
100		10,000
200		20,000

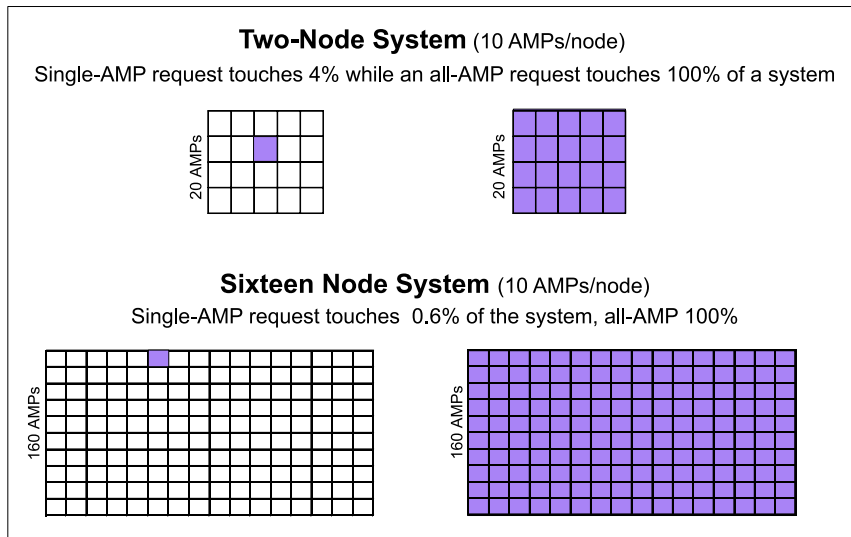
Localizing the Work

After an SQL statement is optimized, the Dispatcher sends the individual steps that make up that query, one at a time, to some number of AMPs in the system. Steps are usually dispatched, and processes started, on all AMPs in the system. Under some conditions, the Optimizer might decide that this query work can be accomplished by accessing only a single AMP in the configuration. This is referred to as a single-AMP operation, or a single-AMP step.

Vantage uses single-AMP steps to localize work to the fewest resources necessary when it is beneficial to do so. This is frequently the case for tactical queries. Single-AMP work frees the other AMPs in the system to do other work, thereby increasing the potential for overall system throughput. This is diagrammed in the following figure:



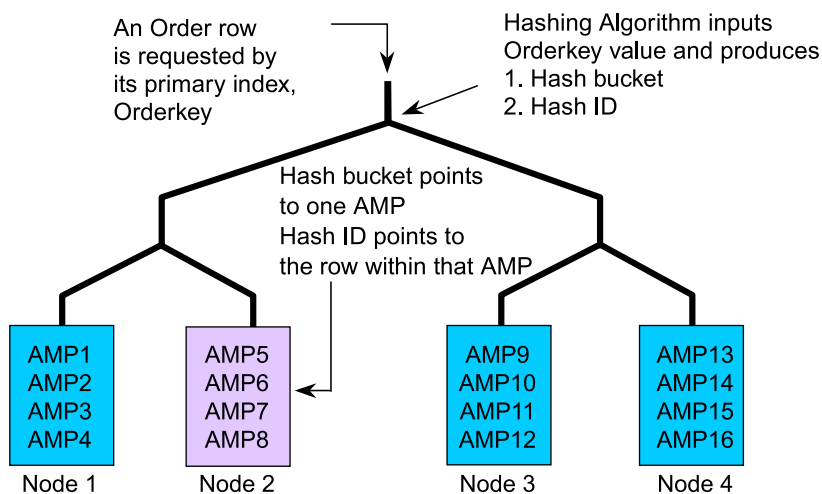
Tactical queries are most efficient and most scalable when designed around single- or few-AMP operations. This is the single most important way to increase throughput and preserve response times for very short tactical queries. This is diagrammed in the following figure:



Single- or few-AMP queries scale well because they engage the same small level of database resources even when the system doubles or triples in size (see [Effect of Configuration Expansion on Tactical Query Response Times](#)).

Single-AMP Operations Using a Primary Index

Single-AMP operations use a primary index value to locate a row.



Single-AMP operations can be achieved by any of the following data manipulation operations:

- Simple single-row inserts
- Simple selects, updates and deletes qualified with a primary index value
- Joins between two tables that share one of the following characteristics:
 - the same primary index domain
 - a primary index making up the join constraint

- a single primary index value is specified for the join

The following EXPLAIN report is for a query that accesses the supplier table using the single primary index value `s_suppkey = 583`. Only a single AMP is engaged, as demonstrated by the single-AMP RETRIEVE step reported in step 1 of the EXPLAIN text:

```
EXPLAIN
SELECT s_name, s_acctbal
FROM supplier
WHERE s_suppkey = 583;
```

Explanation

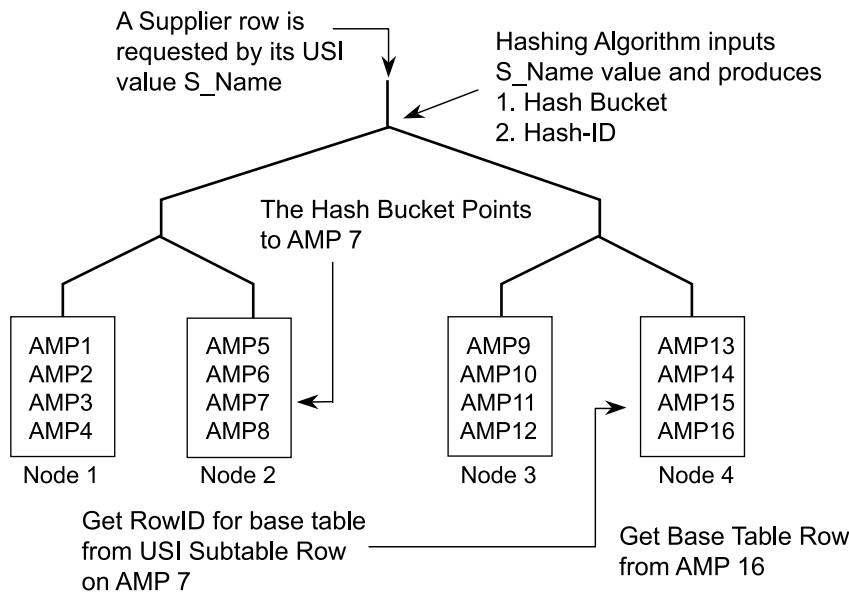
-
- 1) First, we do a **single-AMP RETRIEVE step** from TPCD50G.supplier by way of the unique primary index "TPCD50G.supplier.S_SUPPKEY = 583" with no residual conditions. The estimated time for this step is 0.03 seconds.

Notice that there are no references to locking in the EXPLAIN report for this query. That is because the Optimizer has folded the locking activity (in this case a single row hash READ lock) into the same step that retrieves the row. This sort of lock folding is done only with row hash locks.

This special shortcut for handling row hash locks eliminates the need for the Dispatcher to dispatch a separate locking step when only one AMP and one row are involved. This reduces the PE-to-AMP communication effort.

Two-AMP Operations: USI Access and Tactical Queries

When an application specifies a value that can be used to access a table using its USI, a 2-AMP operation results. Note that USI access can be a single-AMP operation if the USI value for a row happens to hash to a subtable on the same AMP as the primary index for the same row, but is never more than a 2-AMP operation.



In the following query, the column `s_name` is defined as a USI on the original supplier table having `s_supkey` as its UPI:

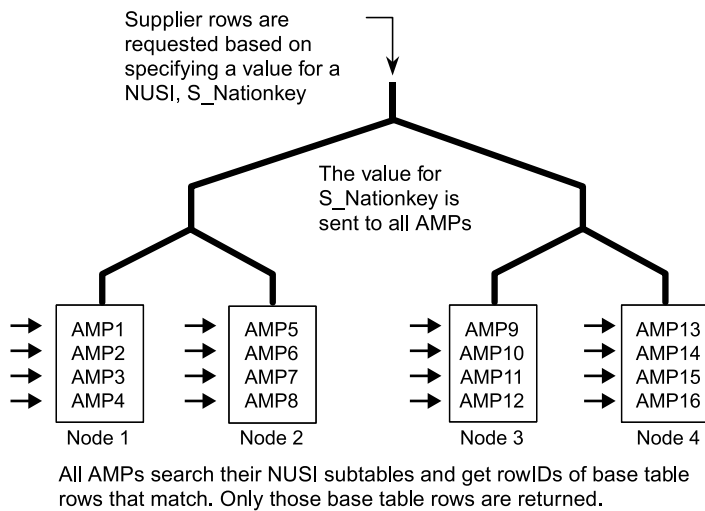
```
EXPLAIN
SELECT s_supkey, s_acctbal
FROM supplier
WHERE s_name = 'Supplier#000038729';
```

Explanation

-
- 1) First, we do a **two-AMP RETRIEVE step** from `CAB.supplier` by way of unique index # 8 "`CAB.supplier.S_NAME = 'Supplier#000038729'`" with no residual conditions. The estimated time for this step is 0.07 seconds.

NUSI Access and Tactical Queries

Compare the previous example of USI access to access using a NUSI. With NUSIs, each AMP contains an index structure for the base table rows that it owns. Even if only a single row contains the specified NUSI value, the Optimizer cannot know that a priori, so the NUSI subtables for all AMPs are always searched, making NUSI accesses all-AMPs operations. The only exceptions are when a NUSI is on the same columns as the primary index (such as when the primary index is row-partitioned) or the NUSI is on the same columns as the primary AMP index. In these cases, only a single-AMP search is required.



The following example assumes the `s_name` column is defined as a NUSI on `supplier`. Notice the difference between this EXPLAIN report and the previous one for the USI (see [Two-AMP Operations: USI Access and Tactical Queries](#)):

```
EXPLAIN
SELECT s_suppkey, s_acctbal
FROM supplier
WHERE s_name = 'SUPPLIER#000000647';
```

Explanation

- 1) First, we lock TPCD50G for read on a RowHash (proxy lock) to prevent global deadlock for TPCD50G.supplier.
- 2) Next, we lock TPCD50G.supplier for read.
- 3) We do an all-AMPs RETRIEVE step from TPCD50G.supplier by way of index # 8 "TPCD50G.supplier.S_NAME = 'SUPPLIER#000000647'" with no residual conditions into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row. The estimated time for this step is 0.20 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

With NUSI access, table-level locks are applied and an all-AMPs operation is performed. Value-ordered indexes and covering indexes are variants of a NUSI. Because of that, they also require all-AMPs operations that are similar to NUSI access.

Group AMP Operations

Sometimes less parallelism is best. Tactical queries, for example, are more efficient when they engage fewer resources. On the other hand, short queries often require more than one AMP and cannot be accommodated with single-AMP processing alone. The Optimizer looks for opportunities to transform what would otherwise be all-AMP query plan into a few-AMP plan. This few-AMPs approach is called Group AMP.

The Group AMP approach not only reduces the number of AMPs active in supporting a query, it also reduces the locking level from a table-level lock to one or more rowkey (specifically, partition and rowhash) locks

in each AMP in the group. Removing the need for table-level locks eliminates two all-AMP steps from the query plan:

- A step to place the table-level lock on all AMPs.
- A step to remove the table-level lock from all AMPs when the query completes.

Determining When the Optimizer Will Consider Group AMP Processing

The Optimizer considers several criteria when it determines whether a query is a candidate for Group AMP processing or not:

- How many AMPs needed to satisfy the request?

For most systems this number of participating AMPs must be 50% or fewer of the total number of AMPs configured in the system. Statistics collected on the selection and join columns being referenced in the query help the Optimizer make the correct assessment. If no statistics for these columns exist, the Group AMP option is less likely to be chosen. This makes the regular collection of statistics a critical prerequisite for gaining the advantages of the Group AMP feature.

- Can the query be driven from a single-AMP or Group AMP step as the first meaningful database access step?

Because NUSI access is always an all-AMP activity, accessing rows by means of a NUSI as the first step in the query plan eliminates Group AMP considerations later in the plan, even if the NUSI access returns only one row from one AMP.

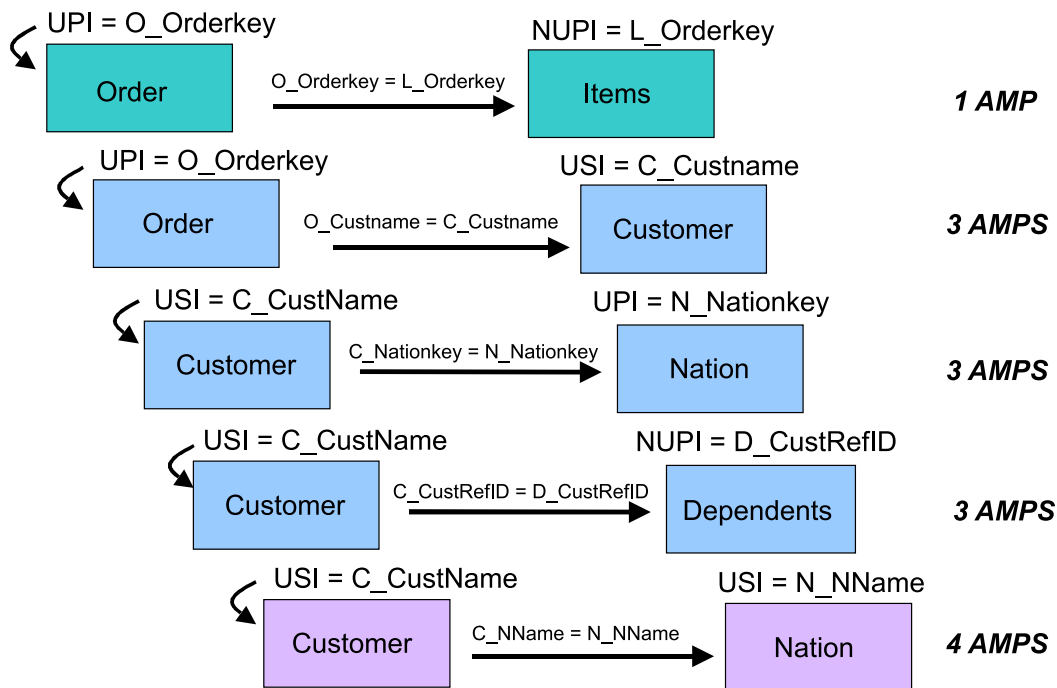
- Does a product join based on table duplication appear in any step in the query plan?

If so, the Group AMP option is not considered. Under this condition, a product join is always an all-AMP operation. The existence of a single all-AMP operation in the plan negates the possibility of using a Group AMP later.

Few-AMP Joins and Tactical Queries

Even when joins must be made to process a tactical query, it is possible to make the join with few-AMP operations. The following picture illustrates the combinations of accessing and joining that can be performed engaging one or few AMPs. In all of these few-AMP examples, the first table is accessed using either a UPI or a USI.

The first join in the following example is made using a merge join. Because the primary index columns of both tables share the same domain, their associated rows reside on the same AMP, with the second table being joined based on a NUPI. The remaining joins use combinations of primary index and USI access and perform few-AMP nested joins between the tables.



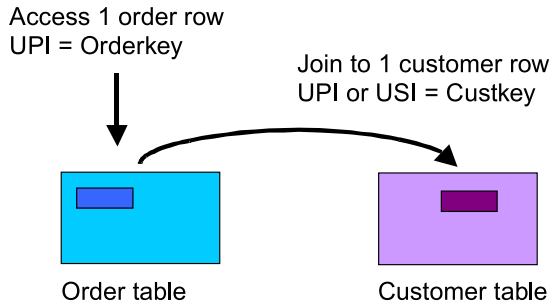
One idiosyncrasy of EXPLAIN text is that it sometimes says “we do a two-AMP join step” when a USI is defined on one half of the nested join. Three AMPs are engaged for most such joins, not two.

Tactical Queries Benefit From Nested Joins

All AMPs are engaged for product and hash joins, and also for most merge joins. For this reason these join methods are less desirable for more localized queries. Tactical queries benefit from the few-AMP joins described in [All-AMP Queries](#).

Of these join methods, the most beneficial for tactical queries is the nested join, which under specific conditions involves only a few AMPs.

A few-AMP nested join is possible when the first table can be accessed by a UPI or USI specified by the request. Following that, a foreign key within that first table must be available to drive a nested join into a second table, which contains its associated primary key. For this nested join to involve only one or a few AMPs, the second table must have either a primary index or a USI defined on the column set mapped to by the foreign key in the first table, as demonstrated by the following graphic:



Database Design Techniques to Support Localized Work

Several approaches to physical database design can make one-AMP or few-AMP query plans more likely. Most of these choices need to be made carefully, because they also influence how other work in the system is optimized. All applications running in the database need to be considered, particularly when selecting or changing primary index columns:

- Use the Same Primary Index Definitions

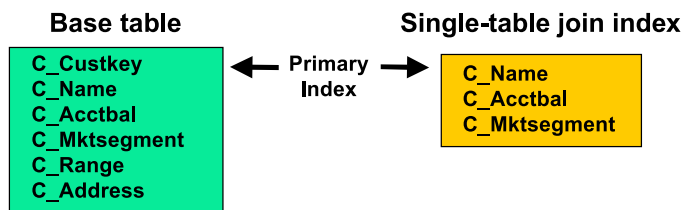
When you expect frequent joins between two associated tables, consider using identical column sets for the primary index definitions of both tables. In other words, define the primary indexes and the join columns are the columns. This technique can enhance both decision support and tactical queries, depending on the frequency of the join and the demographics of the data.

- Increase the Likelihood of Few-AMP Nested Joins

Consider placing USIs on the join constraint of one of the tables. From the perspective of logical database design, you are placing the USI on the primary key of the primary table in the primary key-foreign key relationship.

- Consider a Join Index

You can create a single-table join index or hash index with a different primary index than the base table. For example, you could define a primary index for the join or hash index composed of the column that corresponds to values frequently specified by the application. The example in the following diagram indicates how a query that only has a value for customer name could use the join index for single-AMP access.

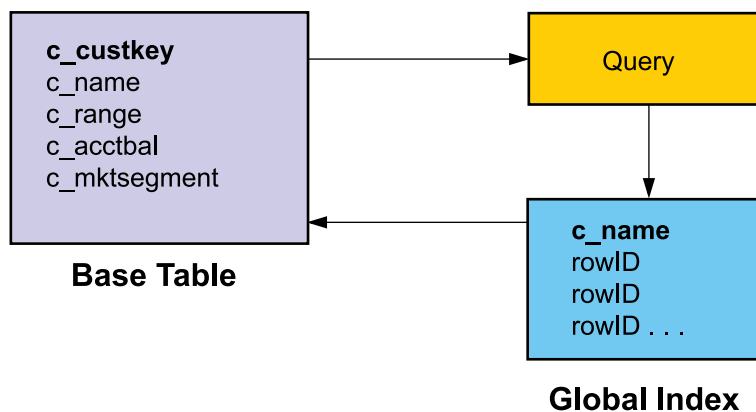


- Consider a Global Join Index

A global join index is a single-table join index, similar to the one illustrated above, except for one important difference: each global join index row contains a pointer to its base table that the Optimizer can use as an alternate way to access base table rows. Using a join index for base table access is referred to as partial covering because such a join index only partially covers the query (see [Partial Query Coverage](#) and [Restrictions on Partial Covering by Join Indexes](#)).

Global join indexes offer the combined advantages of a NUSI (by supporting duplicate rows per value) and a USI (hashed index rows), and the Optimizer can often take advantage of these capabilities for Group AMP operations.

The following graphic indicates how the query first accesses the global join index, then uses the rowID information from the index to access the base table rows:



Suppose you create the following base table:

```

CREATE MULTISET TABLE adw.customer (
  c_custkey    DECIMAL(18,0) NOT NULL,
  c_name       VARCHAR(30) CHARACTER SET LATIN CASESPECIFIC
              NOT NULL,
  c_address    VARCHAR(40) CHARACTER SET LATIN CASESPECIFIC
              NOT NULL,
  c_nationkey  DECIMAL(18,0) NOT NULL,
  c_phone      CHARACTER(10) CHARACTER SET LATIN CASESPECIFIC
              NOT NULL,
  c_mktsegment CHARACTER(10) CHARACTER SET LATIN CASESPECIFIC
              NOT NULL,
  c_comment    CHARACTER(100) CHARACTER SET LATIN CASESPECIFIC
              NOT NULL)
  UNIQUE PRIMARY INDEX ( c_custkey )
  INDEX ( c_nationkey );
  
```

To support group AMP access to the data in this table, you create the following global join index:

```
CREATE JOIN INDEX adw_ji AS
  SELECT (c_phone), (ROWID)
  FROM customer
  PRIMARY INDEX(c_phone);
```

A typical query against the customer table might be something like the following:

```
SELECT c_name, c_mktsegment
FROM customer
WHERE c_phone = '5363333428';
```

To make the Optimizer aware of the opportunity for using group AMP access to respond to this query, you must first collect statistics on the `c_phone` column of the base table. Otherwise, the Optimizer still uses the global join index, but generates an all-AMP plan instead of the more cost effective group AMP plan.

The main advantages of global join indexes are scalability and throughput. Whether a global join index supports faster processing than a NUSI largely depends on how busy the system is at the time the query is submitted. On a system with a light load, a query supported by a NUSI might run slightly faster, because a NUSI scan is one step that all AMPs perform in parallel. To use a global join index, the plan requires two steps to perform the same operation.

When a system is heavily loaded, a global join index is likely to provide a performance advantage, all things being equal, because it does not impel the overhead of an all-AMP operation. The higher the number of AMPs involved in satisfying a request, the higher the likelihood of experiencing resource contention and experiencing response delays, and the difference is more pronounced as a configuration grows in size. As a result, the practical benefits accrued from using a global join index rather than a NUSI also increase.

Because global join indexes can partially cover a query, some, or even most, of the columns requested by a query need not be defined in the join index itself. Multitable join indexes also support partial covering, though aggregate join indexes do not. Frequently only the primary index of the global join index is carried, along with the unique identifier of the base table it supports.

You can specify any of the following unique identifiers in the definition of a global join index:

- The row ID of the base table (expressed as the keyword `ROWID`).
- The primary index of the base table.
- A unique secondary index on the base table.

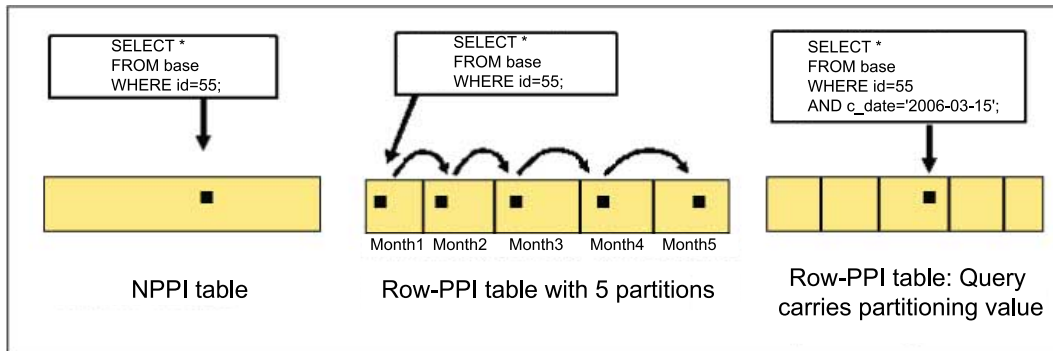
The Optimizer can also use multitable join indexes for partial query coverage capability (see [Partial Query Coverage](#) and [Restrictions on Partial Covering by Join Indexes](#)).

Single-AMP Queries and Partitioned Tables

Primary index row-partitioning refers to the physical ordering of rows within the table. Partitioned rows are grouped on each AMP based on a partitioning key, which can be one or multiple columns (see [Partitioned and Nonpartitioned Primary Indexes](#) for details). The partitioning columns need not be part of the primary

index; however, when the partitioning column is different from the primary index, the performance of tactical queries that pass only a primary index value might be affected.

When the system performs a primary index access to a row-partitioned table and the query does not provide the partitioning key in a condition, each row partition might need to be probed individually to determine if it contains rows reflecting that value. As seen in the following graphic, if you specify a partitioning key value in a WHERE condition (in this case, the condition is `c_date = '2006-03-15'`), the Optimizer can use row partition elimination to probe only the row partition that contains the sought after value.



Probing always occurs when the primary index is nonunique because of the possibility that duplicate primary index rows with different partitioning key values might be spread across the table. The primary index of a partitioned table must be defined as nonunique unless its partitioning key is also included as part of the primary index definition (see [Row-partitioned Primary Indexes](#)). When the partitioning key is included as part of the primary index definition, the system does not need to probe each individual row partition.

Recommendations for Tactical Queries and Row-Partitioned Tables

When executing tactical queries based on primary index access to row-partitioned tables, consider the following suggestions for enhancing query performance, listed in order of greatest general benefit:

1. The best performance is achieved when both of the following conditions are true:
 - The row-partitioned table is defined with a unique primary index.
 - The primary index definition also contains the partitioning columns for the table.

All UPIs on partitioned tables must contain the partitioning columns. Because of this requirement, any primary index access returns a single row, and only a single row partition must be probed to access that row.

2. If you cannot include the partitioning columns in the primary index definition for a row-partitioned table, primary index access will also perform well if you specify a value for the partitioning key WHERE clause as an additional constraint. This specification eliminates the need to probe all of the partitions.
3. You can achieve good performance by defining a NUSI on the NUPI column set for the partitioned table.

4. If for some reason you cannot specify a partitioning key as a condition in a query against a row-partitioned table, you should consider the following alternative methods, where appropriate:

IF you cannot provide the value for the partitioning key in the WHERE clause, the PPI is a NUSI, and its values are ...	THEN consider creating this type of index on the primary index columns of the row-PPI ...
unique	USI.
not unique	global join index.

Either design strategy avoids the necessity of probing each partition because the indexes point directly to each relevant physical row.

5. If none of these methods is appropriate for your workloads, consider defining the row-partitioned table with fewer partitions. By making the row-partition granularity more coarse, you reduce the level of probing required for a primary index access.

Sparse Join Indexes and Tactical Queries

Join indexes are particularly useful for tactical query applications. [Join and Hash Indexes](#) discusses join indexes more fully, but one interesting join index approach that is worth highlighting here for its relevance to tactical queries is the sparse join index.

Sparse join indexes include only a subset of a base table rows in their definition, using a WHERE clause to determine which base tables rows are retained and which are not (see [Sparse Join Indexes](#)). Sparse join indexes are quicker to build, faster to scan, and take up less disk space, depending on the degree of sparseness. Like all join indexes, sparse join indexes support single-AMP access when based on their primary index definition.

Sparse Join Index Defined on One Row Partition

A partitioned primary index can be defined on a join index as long as the index is not row compressed. You can also define a sparse join index on only one row partition of a row-partitioned base table by expressing sparseness-defining criteria that match the borders of the row partition.

Building sparse join indexes on row partitions of row-partitioned tables that support single-AMP access is frequently useful for situations in which tactical queries always have both the sparse-defining column (in this example, a date range that matches one row partition) and the primary index value (in this example, the store identifier) of the sparse join index. Of course, the tactical queries also need to be accessing a similar subset of columns from the base table: the ones carried in the sparse join index.

A different sparse join index could be built independently on several different row partitions of the same row-partitioned table. As long as each query specifies a constraint that matches the sparse-defining columns for one of those sparse join indexes, the Optimizer can choose the appropriate one to use for the query.

The appropriate primary index for a sparse join index depends on what values the tactical queries specify when they are submitted.

Considerations For Using Sparse Join Indexes With Dense NUPIs

Selecting a primary index for a sparse join index that has thousands of rows per value, with each AMP controlling some percentage of these values, provides several benefits and carries few of the negatives associated with a high number of duplicate primary index values.

For example:

- During join index creation there is no duplicate row checking as there is with a base table, so one of the principal reasons to avoid such high numbers of duplicates on a primary index does not apply to the case of creating a join index.
- The join itself can be more efficient with a higher numbers of NUPI duplicates because when so many rows carry the same NUPI row-hash value, the physical I/O involved in storing them can be less.
- While balanced processing is always important when selecting a primary index for a base table, the dense NUPI approach is appropriate for join indexes when it enables fast query execution and replaces an all-AMP alternative that would process only a few rows from each AMP.

On the other hand, it is not desirable to overload one AMP unduly, whether the access is single- or all-AMP. If an inordinate number of data blocks would have to be processed by one AMP using the dense NUPI approach, then parallelizing the work across all AMPs by selecting an alternative primary index is probably a better choice for enhancing performance.

- Designing a sparse join index to ensure that the number of distinct values in its index primary index is greater than the number of AMPs in the system is a good strategy to protect against too many queries being concentrated on too few AMPs. However, if the queries are very short and are infrequent, that concern is less important.

All-AMP Queries

Some tactical queries require all-AMPs steps. All-AMPs queries are likely to have a more relaxed response time expectation than single-AMP queries even when they are capable, at times of low concurrency, of subsecond response.

Coding Suggestions for All-AMP Tactical Queries

All-AMP tactical queries behave differently than few-AMP queries as the level of concurrency increases. Some suggestions for coding tactical queries that involve all-AMPs operations follow:

- Review the queries for unnecessary database access. If found, eliminate those accesses from the query.
- Tune row-partitioned tables so queries against them can avoid unnecessary probing.
- Rely on NUSI access, where possible, to avoid scanning a large table.
- Define a hash or join index that partitions the base table vertically where it makes sense to do so because scanning a join index that contains only a subset of the columns from the base table is faster than scanning the base table.

- Look for possibilities to break complex tactical queries into several smaller statements and encapsulating them within a procedure, particularly when all-AMP operations can be replaced by multiple single-AMP operations.
- Specify explicit ACCESS locks wherever possible (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for more information) or set the default session read lock to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144).

ACCESS locks are more important for all-AMP queries than for single-AMP queries because an all-AMP query can require access to more data over a longer period than a single-AMP query.

All-AMP Tactical Queries and Partitioned Tables

The major benefit that can be realized for all-AMP tactical queries running against partitioned tables is row and column partition elimination. Rather than scanning an entire table, it is possible to scan a smaller subset of rows if the query specifies the partitioning key value for the table or a subset of the columns when column partitioned.

Avoid adjusting the granularity of partitions more finely than is warranted by the majority of the queries that run against the table. For example, if users never access data with a granularity smaller than a monthly range, there is no additional performance advantage to be gained by defining partitions that have a smaller range than one month.

For all-AMP tactical queries, you should include the partitioning columns as a constraint in the WHERE clause of your queries. If a query joins row-partitioned tables that have identical row partitioning, you can enhance join performance still further by including an additional equality constraint on the partitioning columns of the two tables.

Group AMP Check in Final Query Step

Whether a query is single-AMP or all-AMP, simple or very complex, there is always an opportunity to reduce an all-AMP operation to a Group AMP operation.

Group AMP logic restricts which AMPs participate in the BYNET merge activity that is part of response processing. This action eliminates unnecessary all-AMP activities at the end of each query. While this feature is likely to have a greater impact on short, tactical queries, any time all-AMP activities can be eliminated anywhere in any query plan, the potential system throughput increases because resources are freed for other work.

To see how this works, examine the final step in a complex decision support query that returns millions of rows. In this case, because of the number of rows returned, each AMP is active in response processing and the Group AMP group includes them all. For other queries, the group might be a subset of the AMPs that is determined when the step executes.

- 4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (**group_amps**), which is built locally on the AMPs. Then we do a SORT to order SORT to order Spool 1 by the sort key in spool field1. The result spool file will not

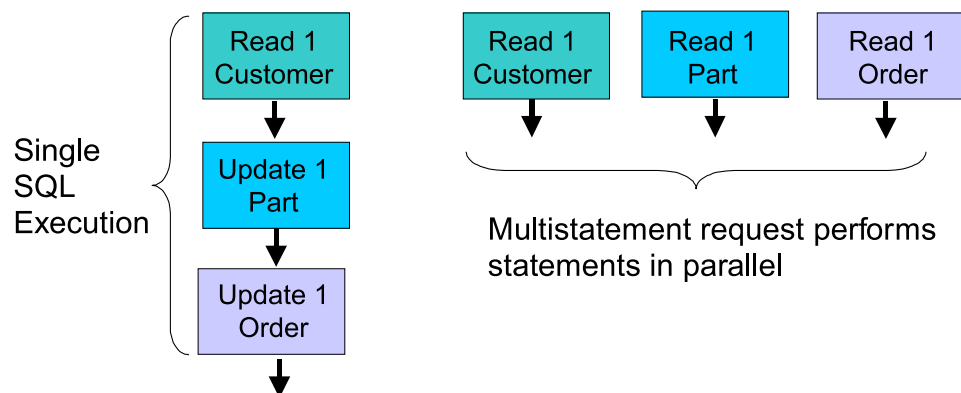
be cached in memory. The size of Spool 1 is estimated with no confidence to be 379,305,309 rows. The estimated time for this step is 32 minutes and 32 seconds.

Application Opportunities for Tactical Queries

In addition to physical database design choices, there are several application opportunities that benefit tactical query performance. The main areas of interest are multistatement requests and macros.

Multistatement Requests

Multistatement requests are a database feature in which multiple SQL statements are bundled together and treated as a single parsing and recovery unit, as illustrated by the following graphic:



Because they are run in parallel, these single-AMP multistatement requests are processed with great efficiency. Multistatement requests are application-independent and can improve performance in a variety of ways. They are particularly useful in improving response times for the combined statements.

The benefits of multistatement requests include the following:

- Communication overhead is reduced
- One parser-optimization process is performed instead of several
- Greater inter-request parallelism is possible

Coding Multistatement Requests

Multistatement requests are application-independent, but their behavior can be different depending on whether they are specified as an implicit transaction or as part of an explicit transaction and whether they are submitted in Teradata or ANSI/ISO session mode.

Vantage treats multistatement requests as single implicit transactions in Teradata session mode when no open BEGIN TRANSACTION statement precedes them.

With the exception of multistatement INSERT requests, Vantage treats multistatement requests as single recovery units only if they are executed either as implicit transactions in Teradata session mode, or in

ANSI/ISO session mode. Depending on any errors generated by the statements in the transaction, either the entire transaction or only the erring request is rolled back. For example, in Teradata session mode within an explicit transaction, Vantage rolls the entire transaction back to the BEGIN TRANSACTION statement, so in this case, the complete transaction is the recovery unit.

If several UPDATE requests are specified within one multistatement request inside an explicit transaction in Teradata session mode, and one of them fails, then all are rolled back. The fewer UPDATE statements in the multistatement request, the lower the impact of the rollback. However, the more statements included in the request, the higher the degree of parallelism among them.

The rollback issue for UPDATE requests is not true for multistatement INSERT requests, where the statement independence feature can frequently enable multistatement INSERT requests to roll back only the statements that fail within an explicit transaction or multistatement request and not the entire transaction or request.

Statement independence supports the following multistatement INSERT data error types:

- Column-level CHECK constraint violations
- Data translation errors
- Duplicate row errors for SET tables
- Primary index uniqueness violations
- Referential integrity violations
- Secondary index uniqueness violations

Statement independence is not enabled for multistatement INSERT requests into tables defined with the following options:

- Triggers
- Hash indexes
- Join indexes

See the information about INSERT and INSERT SELECT in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about statement independence. Note that various client data loading utilities also support statement independence. Consult the appropriate Teradata Tools and Utilities documentation for information about which load utilities and APIs support statement independence and what level of support they offer for the feature.

ACCESS Locking and Multistatement Requests

If you want each statement in a multistatement request to use row hash-level ACCESS locking, specify the LOCKING ROW FOR ACCESS modifier preceding each individual SELECT statement in the multistatement request. See [Coding Multistatement Requests](#) for additional details about how Vantage handles transactions, particularly those containing multistatement requests.

Macros and Tactical Queries

You automatically get a multistatement request without coding it if you write multiple SQL statements within a macro because macros are always performed as a single request. The macro code can also specify parameters that are replaced by data each time the macro is performed. The most common way of

substituting for the parameters is to specify a USING request modifier (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

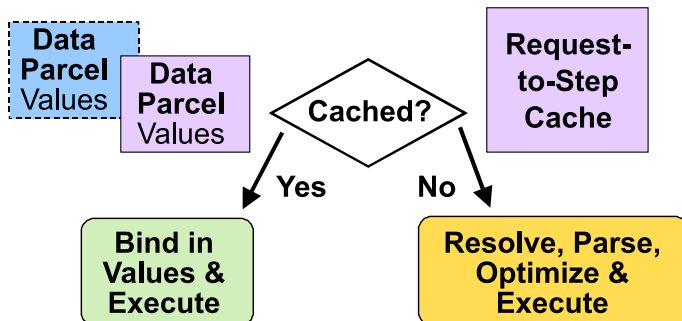
Macros are usually more efficient for repetitive queries than single DML statements. Unlike procedures, macros are not designed to return parameters to the requestor: they return only the answer set for the SQL statements they contain.

Macros have the following advantages:

- Network and channel traffic are reduced.
- Execution plans can be cached, reducing parsing engine overhead.
- They ensure the efficient performance of their component SQL statements.
- Reusable database components save client resources.
- They can be used to force data integrity and locking modifiers.
- They can be used to enforce security.

Cached Plans

Caching of repeatable requests in the Request-to-Step Cache reduces performance time because parsing and optimizing do not need to be done when cached requests are repeated. For subsecond queries, cached plans significantly reduce the query time and enhance throughput. The following figure is a high-level flow chart for the request-to-steps cache that resides in each parsing engine on a Teradata system.



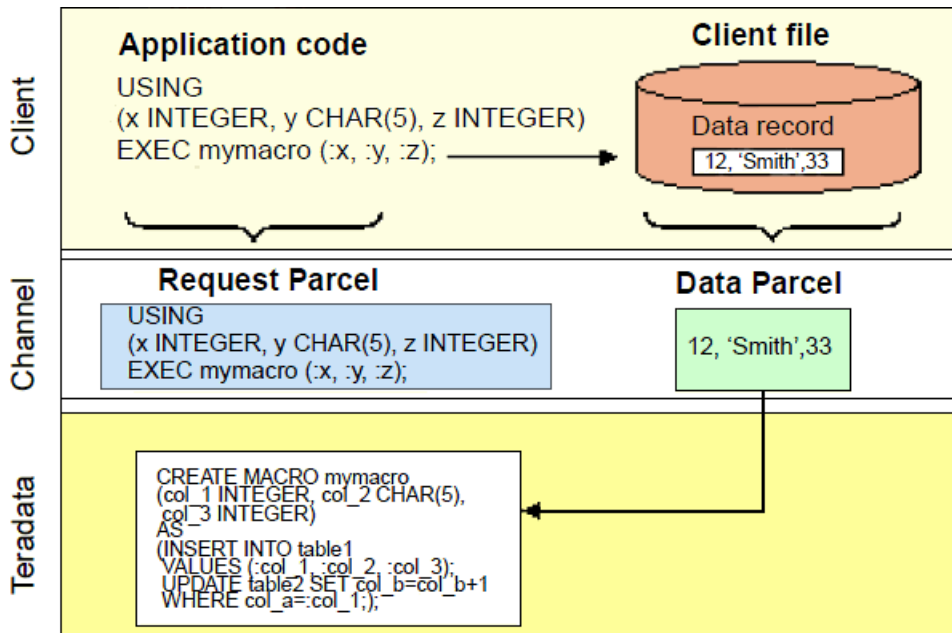
For macros to be considered for caching, Vantage knows that the SQL text they contain is repeatable. All of the following attributes must be identical each time the request is sent to the Teradata platform:

- Host, workstation, or LAN type
- Default database name
- National character set
- Request text

Parameter variables in the macro support repeatability. These parameters must be specified in both the SQL code itself and in the USING request modifier that precedes the EXEC statement that invokes the macro.

The following picture illustrates the conditions required for caching the SQL statements for a macro. In this example, the request is made using BTEQ from a mainframe client across a block multiplexor channel. Note

that the SQL code is in a request parcel, while the data values are transmitted in a data parcel. Both a request and a data parcel must be transmitted for the SQL to be cached immediately.



The data values are passed from the application to the macro code where they are then processed. Their format depends on the programming conventions for the application. For example, in BTEQ you must use the `.IMPORT FILE =` command when you perform the macro, and the import file specified in the command contains records with the data values to be passed to the macro. The values in the file are positionally associated, from left to right, with the parameters in the `USING` request modifier.

Other Tools Useful for Monitoring and Managing Tactical Queries

The following recommendations refer to monitoring and management facilities other than Database Query Logging or Teradata Viewpoint that might be active in a Teradata data warehouse. These recommendations are made with tactical query applications in mind; however, their value should be understood in light of the entire Teradata environment.

- Avoid using the account string expansion &T variable for tactical queries, because it is likely to write one row in AmpUsage for every request issued. This extra processing overhead can impact response time and reduces the throughput of single- or few-AMP tactical queries. See *Teradata Vantage™ - Database Administration*, B035-1093 for more information about account string expansion variables.
- Stop using Access Logging to log queries. Database Query Logging provides most of the same functionality and offers much more. DBQL also has considerably less impact on the performance of tactical query applications. See *Teradata Vantage™ - Database Administration*, B035-1093 and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about Access Logging and Query Logging.

- Taking frequent snapshots using Priority Scheduler monitor (every 5 or 10 seconds) incurs no meaningful overhead and does not affect tactical query performance. Information from this monitor output can be useful in understanding how priorities are functioning and which groups are using what extent of CPU activity.
- A ten-minute collection interval for ResUsage has proven sufficient in many diverse environments, and minimizes any impact on tactical queries. ResUsage data is critical to tracking overall system-level health and usage patterns, and for capacity planning. See *Teradata Vantage™ - Resource Usage Macros and Tables*, B035-1099 for more information about ResUsage.

Monitoring Active Work

Workload management is critical to the success of tactical query applications. Improved workload management starts with understanding what is active on the platform and what the resource consumption profiles of your different applications look like.

Recommended Monitoring Activities

A quick list of monitoring activities that offer a foundation for workload management follows:

- Use Teradata Viewpoint to monitor and manage the workload.
- Collect user resource usage detail data.

The DBC.Acctg table is the underlying table for the AMPUsage view, and captures data about user usage of CPU and I/O for each AMP. Heavy resource consumers over time, skewed work, and application usage trends can be identified.

- Collect ResUsage data.

The ResUsage tables report on the aggregate effect of all user requests on various system components over time, and can identify bottlenecks and capacity issues. See *Teradata Vantage™ - Resource Usage Macros and Tables*, B035-1099 for details on all ResUsage data and macros.

- Use the Lock Viewer Viewpoint portlet.

Lock Viewer is essential for identifying locking conflicts.

- Use Database query logging (DBQL).

The Database Query Log records details on queries run, including arrival rates, response times, objects accessed, and SQL statements performed. See *Teradata Vantage™ - Database Administration*, B035-1093 and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about Query Logging.

- Enable canary queries.

You should use canary queries with tactical query applications. See the following section for more information about this monitoring technique.

Using Canary Queries

Canary queries are SQL statements that represent the characteristics of a particular application. These queries are introduced into the work stream entering your data warehouse in order to monitor system responsiveness. Some sites run canary queries once every 1 to 5 minutes, while others run them only every 30 minutes. Canary queries provide a quick health check for tactical applications specifically and for your Teradata system in general. By monitoring the response times when the canary queries run, you can identify delays or congestion states early on.

Some users run canary queries in each workload that is active, others only use them for their tactical query applications or in workloads that have a defined service level.

Many sites make charts or graphs of canary query behavior during the previous 24 hours, with particular attention paid to any outliers. When the response time for a canary query is out of line with expectations, you can attempt to match that occurrence to system conditions at that time. It is a common practice for canary queries that exceed a specified response time threshold to send an e-mail alert to the DBA staff.

Take care to avoid over-scheduling canary queries. If the information your canary queries produce is too large to be easily processed and analyzed, reduce the scope of their execution.

Teradata Viewpoint is designed to automate the submission of canary queries. See the Teradata Viewpoint online HELP to learn how to do this.

Notation Conventions

This section describes the notation conventions used in this document.

Throughout this document, two conventions are used to describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

For example:

- DECIMAL [(n[,m])] means the decimal data type can be defined optionally:
 - without specifying the precision value *n* or scale value *m*
 - specifying precision (*n*) only
 - specifying both values (*n*, *m*)
 - you cannot specify scale without first defining precision.
- CHARACTER [(*n*)] means that use of (*n*) is optional.

The values for *n* and *m* are integers in all cases.

Symbols from the predicate calculus, set theory, and dependency theory are also used occasionally to describe logical operations. See [Predicate Calculus and Set Theory Notation Used in This Document](#) and [Dependency Theory Notation Used in This Document](#).

Table Column Definition and Constraint Abbreviations

The following set of abbreviations is used throughout this document to denote various characteristics of, or constraints on, table columns.

Abbreviation	Definition
FK	Foreign key
NC	No changes allowed
ND	No duplicates allowed
NN	No nulls allowed
PK	Primary key
SA	System-assigned value
UA	User-assigned value

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a–z A–Z 0–9	Any	Any single byte Latin letter or digit.
a–z A–Z 0–9	Any	Any fullwidth Latin letter or digit.
<	KanjiEUC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEUC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by “ss 3”.
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by “ss 2”, forming an individual multibyte character.
Δ	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as TEST. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Predicate Calculus and Set Theory Notation Used in This Document

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this document uses the following predicate calculus and set theory notation to explain concepts.

This symbol ...	Represents this phrase ...
Predicate Calculus Notation	
iff	If and only if
\forall	For all
\exists	There exists
Set Theory Notation	
\emptyset	Empty set
\cup	Union
\cap	Intersection
\aleph	Infinitely large set Pronounced "aleph"

Dependency Theory Notation Used in This Document

The decomposition of relations during normalization relies on the notation of dependency theory.

Occasionally this uses the following dependency theory notation to explain concepts:

This symbol ...	Represents this phrase ...
\rightarrow	Functionally determines.
\nrightarrow	Does not functionally determine.
\Rightarrow	Multivalued determines.

Teradata System Limits

This section provides the following limits.

- System limits
- Database limits
- Session limits
- System-derived and system-generated column data types

The reported limits apply only to the platform software. Platform-dependent client limits are not documented.

System Limits

The system specifications in the following tables apply to an entire Teradata Vantage configuration.

Miscellaneous System Limits

Parameter	Value
Maximum number of combined databases, users, and zones.	4.2 x 10 ⁹
Maximum number of database objects per system lifetime. A database object is any object whose definition is recorded in DBC.TVM. Because the system does not reuse DBC.TVM IDs, this means that a maximum of 1,073,741,824 such objects can be created over the lifetime of any given system. At the rate of creating one new database object per minute, it would take 2,042 years to use 1,073,741,824 unique IDs.	1,073,741,824
Maximum size for a table header.	1 MB
Maximum size of table header cache.	8 x 10 ⁶ bytes
Maximum size of a response spool row.	<ul style="list-style-type: none"> • 1 MB on large-cylinder systems • 64 KB on small-cylinder systems
Maximum number of change logs that can be specified for a system at any point in time.	1,000,000
Maximum number of locks that can be placed on aggregate online archive logging tables, databases, or both per request. The maximum number of locks that can be placed per LOGGING ONLINE ARCHIVE ON request is fixed at 25,000 and cannot be altered.	25,000
Maximum number of 64KB parse tree segments allocated for parsing requests.	12,000
Maximum number of nodes per system configuration.	1,024

Parameter	Value
Limits on vprocs of each type restrict systems with a large number of nodes to fewer vprocs per node. Systems with the maximum vprocs per node cannot approach the maximum number of nodes.	
Maximum size of a query band.	4,096 UNICODE characters

Message Limits

Parameter	Value
Maximum number of CLlv2 parcels per message	256
Maximum request message size (client-to-database)	7 MB This limit applies to messages to and from client systems and to some internal Vantage messages.
Maximum response message size (database-to-client)	16 MB
Maximum error message text size in a failure parcel	255 bytes

Storage Limits

Parameter	Value
Total data capacity	~ 12 TB/AMP
Minimum data block size with small cylinders	~9KB 18 512-byte sectors
Default data block size with small cylinders	~127KB 254 512-byte sectors
Maximum data block size with small cylinders	~256KB 512 512-byte sectors
Minimum data block size with large cylinders	~21KB 42 512-byte sectors
Default data block size with large cylinders	~127KB 254 512-byte sectors
Maximum data block size for a large cylinder system that does not use 4KB alignment	1,023.5KB 2,047 sectors
Maximum data block size for a large cylinder system that uses 4KB alignment	1,024KB 2,048 sectors

Parameter	Value
Maximum number of data blocks that can be merged per data block merge	8
Maximum merge block ratio block size for a table.	100% of the maximum multirow block size for a table.

Gateway and Vproc Limits

Parameter	Value
Maximum number of sessions per PE.	Multiple. This is true because the gateway runs in its own vproc on each node. See <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 for details. The exact number depends on whether the gateways belong to different host groups and listen on different IP addresses.
Maximum number of sessions per gateway.	Tunable: 1 - 2,147,483,647. 1,200 maximum certified. The default is 600. See <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 for details.
	30,720 This includes the sum of all of the following types of vproc for a configuration: <ul style="list-style-type: none"> • AMP Access Module Processor vprocs • GTW Gateway Control vprocs • PE Parsing Engine vprocs • RSG Relay Services Gateway vprocs • TVS Teradata Virtual
Maximum number of AMP vprocs per system.	16,200
Maximum number of GTW vprocs per system. This is a soft limit that Teradata Services personnel can reconfigure for you. Each GTW vproc on a node must be in a different host group. If two GTW vprocs are in the same host group, they must be on different nodes.	The default is one GTW vproc per node with all of them assigned to the same host group. The maximum depends on: <ul style="list-style-type: none"> • Number of IP addresses assigned to each node.

Parameter	Value
	<ul style="list-style-type: none"> Number of host groups configured in the database. <p>Each gateway on a node must be assigned to a different host group from any other gateway on the same node and each gateway needs to be assigned a disjoint set of IP addresses to service. This does not mean that all gateways in a system must be assigned to a different host group. It means that each gateway on the same node must be assigned to a different host group. Gateways on different nodes can be assigned to the same host group.</p>
Maximum number of PE vprocs per system. This is a soft limit that Teradata Services personnel can reconfigure for you.	2,048
Maximum number of TVS allocator vprocs per system. This is a soft limit that Teradata Services personnel can reconfigure for you.	2,048
Maximum number of vprocs, in any combination, per node.	127
Maximum number of AMP vprocs per cluster.	8
Maximum number of external routine protected mode platform tasks per PE or AMP. This value is derived by subtracting 1 from the maximum total of PE and AMP vprocs per system (because each system must have at least one PE), which is 16,384. This is obviously not a practical configuration. The valid range is 0 to 20, inclusive. The limit is 20 platform tasks for each platform type, not 20 combined for both. See <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 for details.	20
Maximum number of external routine secure mode platform tasks per PE or AMP. This value is derived by subtracting 1 from the maximum total of PE and AMP vprocs per system (because each system must have at least one PE), which is 16,384. This is obviously not a practical configuration.	20
Size of a request control block	~ 40 bytes
Default number of lock segments per AMP vproc. This is controlled by the NumLokSegs parameter in DBS Control.	2
Maximum number of lock segments per AMP vproc. This is controlled by the NumLokSegs parameter in DBS Control.	8

Parameter	Value
Default size of a lock segment. This is controlled by the LockLogSegmentSize parameter in DBS Control.	64 KB
Maximum size of a lock segment. This is controlled by the LockLogSegmentSize parameter in DBS Control.	1 MB
Default number of locks per AMP	3,200
Maximum number of locks per AMP.	209,000
Maximum size of the lock table per AMP. The AMP lock table size is fixed at 2 MB and cannot be altered.	2 MB
Maximum size of the queue table FIFO runtime cache per PE.	<ul style="list-style-type: none"> • 100 queue table entries • 1 MB
Maximum number of SELECT AND CONSUME requests that can be in a delayed state per PE.	24
Amount of private disk swap space required per protected or secure mode server for C/C++ external routines per PE or AMP vproc.	256 KB
Amount of private disk swap space required per protected or secure mode server for Java external routines per node.	30 MB

Hash Bucket Limits

Parameter	Value
Number of hash buckets per system. This value is user-selectable.	<p>The number is user-selectable per system. The choices are:</p> <ul style="list-style-type: none"> • 65,536 • 1,048,576 <p>Bucket numbers range from 0 to the system maximum.</p>
Size of a hash bucket	<p>The size depends on the number of hash buckets on the system if the system has:</p> <ul style="list-style-type: none"> • 65,536 hash buckets, the size of a hash bucket is 16 bits. • 1,048,576 hash buckets, the size of a hash bucket is 20 bits. <p>Set the default hash bucket size for your system using the DBS Control utility (see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102 for details).</p>

Interval Histogram Limits

Parameter	Value
Number of hash values.	4.2 x 10 ⁹
Maximum number of intervals per index or column set histogram. The system-wide maximum number of interval histograms is set using the MaxStatsInterval parameter of the DBS Control record.	500
Default number of intervals per index or column set histogram.	250
Maximum number of equal-height intervals per interval histogram.	500

Database Limits

The database specifications in the following tables apply to a single database. The values presented are for their respective parameters individually and not in combination.

Name and Title Size Limits

Parameter	Value
Maximum name size for database objects, for example, account, attribute, authorization, column, constraint, database, function, GLOP, index, macro, method, parameter, password, plan directive, procedure, profile, proxy user, query, role, procedure, table, transform group, trigger, UDF, UDM, UDT, using variable name, view. See <i>Teradata Vantage™ - SQL Fundamentals</i> , B035-1141 for other applicable naming rules.	128 UNICODE characters
Maximum system name size. Used in SQL statements for target level emulation. See <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.	63 characters
Maximum SQL text title size.	256 characters

User Limits

Parameter	Value
Maximum number of external roles.	50

Table and View Limits

Parameter	Value
Maximum number of journal tables per database.	1
Maximum number of error tables per base data table.	1

Parameter	Value
Maximum number of columns per base data table or view.	2,048
Maximum number of columns per error table. This limit includes 2,048 data table columns plus 13 error table columns.	2,061
Maximum number of UDT columns per base data table. The same limit is true for both distinct and structured UDTs. The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space.	~1,600
Maximum number of LOB and XML columns per base data table. This limit includes predefined type LOB columns and UDT LOB columns. A LOB UDT or XML UDT column counts as one column even if the UDT is a structured type that has multiple LOB attributes.	32
Maximum number of columns created over the life of a base data table.	2,560
Maximum number of rows per base data table.	Limited only by disk capacity.
Maximum number of bytes per table header per AMP. A table header that is large enough to require more than ~64,000 bytes uses multiple 64KB rows per AMP up to 1 MB. A table header that requires 64,000 or fewer bytes uses only a single row and does not use the additional rows that are required to contain a larger table header. The maximum size for a table header is 1 MB.	1 MB
Maximum number of characters per SQL index constraint.	16,000
Maximum non-spool row size.	1 MB
Maximum internal spool row size.	~ 1MB
Maximum size of the queue table FIFO runtime cache per table.	2,211 row entries
Maximum logical row size. In this case, a logical row is defined as a base table row plus the sum of the bytes stored in a LOB or XML subtable for that row.	67,106,816,000 bytes
Maximum non-LOB column size for a nonpartitioned table. This limit is based on subtracting the minimum row overhead value for a nonpartitioned table row (12 bytes) from the system-defined maximum row length (64,256 bytes).	64,244 bytes
Maximum non-LOB column size for a partitioned table. This limit is based on subtracting the minimum row overhead value for a partitioned table row (16 bytes) from the system-defined maximum row length (64,256 bytes).	64,240 bytes
Maximum number of values (excluding NULLs) that can be multivalued compressed per base table column.	255
Maximum amount of BYTE data per column that can be multivalued compressed.	4,093 bytes

Parameter	Value
Maximum amount of data per column that can be multivalue compressed for GRAPHIC, LATIN, KanjiSJIS, and UNICODE server character sets.	8,188 characters
<p>Maximum number of columns that can be compressed per primary-indexed or primary-AMP-indexed table or join index using multivalue compression or algorithmic compression.</p> <p>This assumes that the object is not a NoPI table or join index or a global temporary trace table. All other tables, hash indexes, and join indexes must have a primary index or a primary AMP index. Primary indexes and primary AMP indexes cannot be either multivalue compressed or algorithmically compressed. Because of this, the limit is the maximum number of columns that can be defined for a table, which is 2,048, minus 1.</p> <p>The limit for multivalue compression is far more likely to be reached because of table header overflow, but the amount of table header space that is available for multivalue compressed values is limited by a number of different factors.</p> <p>Join index columns inherit their compression characteristics from their parent tables.</p>	2,047
Maximum number of columns that can be compressed per NoPI table using multivalue compression or algorithmic compression. Column-partitioned NoPI join index columns inherit their compression characteristics from their parent tables.	2,048
Maximum number of algorithmically compressed values per base table column.	Unlimited
Maximum width of data that can be multivalue compressed for BYTE, BYTEINT, CHARACTER, GRAPHIC, VARCHAR, and VARGRAPHIC data types.	Unlimited
Maximum width of data that can be multivalue compressed for data types other than BYTE, BYTEINT, CHARACTER, DATE, GRAPHIC, VARCHAR, and VARGRAPHIC.	Unlimited
<p>Maximum width of data that can be algorithmically compressed.</p> <p>The maximum data width is unlimited if you specify only algorithmic compression for a column.</p> <p>If you specify a mix of multivalue and algorithmic compression for a column, then the limits for multivalue compression also apply for algorithmic compression.</p>	Unlimited
Maximum number of table-level CHECK constraints that can be defined per table.	100
Maximum number of primary indexes or primary AMP indexes per table, hash index, or join index that is not a NoPI database object.	1
Minimum number of primary indexes per primary-indexed table, hash index, or join index.	1
Minimum number of primary AMP indexes per primary-AMP-indexed table or join index.	1
Maximum number of primary indexes per primary-AMP-indexed or NoPI or join index or global temporary trace table.	0
Maximum number of columns per primary index or primary AMP index.	64
Maximum number of column partitions per table, including two columns partitions reserved for internal use).	2,050

Parameter	Value
Minimum number of column partition numbers that must be available for use by an ALTER TABLE request to alter a column partition.	1
Maximum partition number for a column partitioning level. Maximum number of column partitions for that level + 1	Maximum number of column partitions for that level + 1
Maximum combined partition number for a single-level column-partitioned table or column-partitioned join index.	The same as the maximum partition number for the single partitioning level.
Maximum number of rows per hash bucket for a 44-bit uniqueness value.	17,592,186,044,415
Maximum combined partition number for a multilevel partitioning for 2-byte partitioning.	65,535
Maximum combined partition number for a multilevel partitioning join index for 8-byte partitioning.	9,223,372,036,854,775,807
Maximum number of ranges, excluding the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 2-byte partitioning. This value is limited by the largest possible INTEGER value.	65,533
Maximum number of ranges, excluding the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 8-byte partitioning. This value is limited by the largest possible BIGINT value.	9,223,372,036,854,775,805
Minimum value for <i>n</i> in a RANGE#L <i>n</i> expression.	1
Minimum value for <i>n</i> in a RANGE#L <i>n</i> expression for 2-byte partitioning.	15
Minimum value for <i>n</i> in a RANGE#L <i>n</i> expression for 8-byte partitioning.	62
Maximum number of partitions, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN partitions, for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type.	2.147.483.647
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	65,533
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	9,223,372,036,854,775,805
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if both the NO RANGE and UNKNOWN partitions are specified.	9,223,372,036,854,775,807
Maximum value for a partitioning expression that is not based on a RANGE_N or CASE_N function. This is allowed only for single-level partitioning.	65,535

Parameter	Value
Maximum number of defined partitions for a column partitioning level.	The number of column partitions specified + 2. The 2 additional partitions are reserved for internal use.
Maximum number of defined partitions for a row partitioning level if the row partitions specify the RANGE_N or CASE_N function.	The number of row partitions specified.
Maximum number of defined partitions for a row partitioning level if the row partitions do not specify the RANGE_N or CASE_N function.	65,535
Maximum number of partitions for a partitioning level when you specify an ADD clause. This value is computed by adding the number of defined partitions for the level plus the value of the integer constant specified in the ADD clause.	9,223,372,036,854,775,807
Maximum number of partitions for a column partitioning level when you do not specify an ADD clause and at least one row partitioning level does not specify an ADD clause.	The number of column partitions defined + 10.
Maximum number of column partitions for a column partitioning level when you do not specify an ADD clause, you also specify row partitioning, and each of the row partitions specifies an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum number of partitions for each row partitioning level without an ADD clause in level order, if using the number of row partitions defined as the maximum for this and any lower row partitioning level without an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum partition number for a row-partitioning level.	The same as the maximum number of partitions for the level.
Minimum number of partitions for a row-partitioning level.	2
Maximum number of partitions for a CASE_N partitioning expression. This value is limited by the largest possible INTEGER value.	2,147,483,647
Maximum value for a RANGE_N function with an INTEGER data type.	2,147,483,647
Maximum value for a RANGE_N function with a BIGINT data type that is part of a partitioning expression.	9,223,372,036,854,775,805
Maximum value for a CASE_N function for both 2-byte and 8-byte partitioning.	2,147,483,647
Maximum number of partitioning levels for 2-byte partitioning.	15

Parameter	Value
Other limits can further restrict the number of levels for a specific partitioning.	
Maximum number of partitioning levels for 8-byte partitioning. Other limits can further restrict the number of levels for a specific partitioning.	62
Maximum value for n for the system-derived column PARTITION#L n .	62
Minimum number of partitions per row-partitioning level for a multilevel partitioning primary index.	1
Minimum number of partitions defined for a row-partitioning level.	2 or greater
Maximum number of partition number ranges from each level that are not eliminated for static row partition elimination for an 8-byte row-partitioned table or join index.	8,000
Maximum number of table-level constraints per base data table.	100
Maximum size of the SQL text for a table-level index CHECK constraint definition.	16,000 characters
Maximum number of referential integrity constraints per base data table.	64
Maximum number of columns per foreign and parent keys in a referential integrity relationship.	64
Maximum number of characters per string constant.	31,000
Maximum number of row-level security constraints per table, user, or profile.	5
Maximum number of row-level security statement-action UDFs that can be defined per table.	4
Maximum number of non-set row-level security constraint encodings that can be defined per constraint. The valid range is from 1 to 10,000. 0 is not a valid non-set constraint encoding.	10,000
Maximum number of set row-level security constraint encodings that can be defined per constraint.	256

Spool Space Limits

Parameter	Value
Maximum internal spool row size.	~ 1MB

BLOB, CLOB, XML, and Related Limits

Parameter	Value
Maximum BLOB object size	2,097,088,000 8-bit bytes
Maximum CLOB object size	<ul style="list-style-type: none"> 2,097,088,000 single-byte characters

Parameter	Value
	<ul style="list-style-type: none"> 1,048,544,000 double-byte characters
Maximum XML object size	2,097,088,000 8-bit bytes
Maximum number of LOB rows per rowkey per AMP for NoPI LOB or XML tables	~ 256M The exact number is 268,435,455 LOB or XML rows per rowkey per AMP.
Maximum size of the file name passed to the AS DEFERRED BY NAME option in a USING request modifier	VARCHAR(1024)

User-Defined Data Type, ARRAY Data Type, and VARRAY Data Type Limits

Parameter	Value
Maximum structured UDT size. This value is based on a table having a 1 byte (BYTEINT) primary index. Because a UDT column cannot be part of any index definition, there must be at least one non-UDT column in the table for its primary index. Row header overhead consumes 14 bytes in an NPPI table and 16 bytes in a PPI table, so the maximum structured UDT size is derived by subtracting 15 bytes (for an NPPI table) or 17 bytes (for a PPI table) from the row maximum of 64,256 bytes.	<ul style="list-style-type: none"> 64,242 bytes (NoPI table) 64,241 bytes (NPPI table) 64,239 bytes (PPI table)
Maximum number of UDT columns per base data table. The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space. The figure of 1,600 UDT columns assumes a FAT table header. This limit is true whether the UDT is a distinct or a structured type.	~1,600
Maximum database, user, base table, view, macro, index, trigger, procedure, UDF, UDM, UDT, constraint, or column name size. Other rules apply for Japanese character sets, which might restrict names to fewer than 30 bytes. See <i>Teradata Vantage™ - SQL Fundamentals</i> , B035-1141 for the applicable rules.	30 bytes in Latin or Kanji1 internal representation
Maximum number of attributes that can be specified for a structured UDT per CREATE TYPE or ALTER TYPE request. The maximum is platform-dependent, not absolute.	300 - 512
Maximum number of attributes that can be defined for a structured UDT. While you can specify no more than 300 to 512 attributes for a structured UDT per CREATE TYPE or ALTER TYPE request, you can submit any number of ALTER TYPE requests with the ADD ATTRIBUTE option specified as necessary to add additional attributes to the type up to the upper limit of approximately 4,000.	~4,000
Maximum number of levels of nesting of attributes that can be specified for a structured UDT.	512
Maximum number of methods associated with a UDT.	~500

Parameter	Value
There is no absolute limit on the number of methods that can be associated with a given UDT. Methods can have a variable number of parameters, and the number of parameters directly affects the limit, which is due to Parser memory restrictions. There is a workaround for this issue. See ALTER TYPE information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184 for details.	
Maximum number of input parameters with a UDT data type of VARIANT_TYPE that can be declared for a UDF definition.	8
Minimum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	2
Maximum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	5

Macro, UDF, SQL Procedure, and External Routine Limits

Parameter	Value
Maximum number of parameters specified in a macro.	2,048
Maximum expanded text size for macros and views.	2 MB
Maximum number of open cursors per procedure.	15
Maximum number of result sets a procedure can return.	15
Maximum number of columns returned by a dynamic result table function. The valid range is from 1 to 2,048. There is no default.	2,048
Maximum number of dynamic SQL requests per procedure.	15
Maximum length of a dynamic SQL request in a procedure. This includes its SQL text, the USING data (if any), and the CLIV2 parcel overhead.	Approximately 1 MB
Maximum combined size of the parameters for a procedure	1 MB for input parameters 1 MB for output (and input/output) parameters
Maximum size of condition names and UDF names specified in a procedure.	30 bytes
Maximum number of parameters specified in a UDF defined without dynamic UDT parameters.	128
Maximum number of parameters that can be defined for a constructor method for all types except ARRAY/VARRAY	128
Maximum number of parameters that can be defined for a constructor method of an ARRAY/VARRAY type	<i>n</i>

Parameter	Value
	where n is the number of elements defined for the type.
Maximum number of combined return values and local variables that can be declared in a single UDF.	Unlimited
Maximum number of combined external routine return values and local variables that can be instantiated at the same time per session.	1,000
Maximum combined size of the parameters defined for a UDF.	1 MB for input parameters 1 MB for output parameters
Maximum number of parameters specified in a UDF defined with dynamic UDT parameters. The valid range is from 0 to 15. The default is 0.	1,144
Maximum number of parameters specified in a method.	128
Maximum number of parameters specified in an SQL procedure.	256
Maximum number of parameters specified in an external procedure written in C or C++.	256
Maximum number of parameters specified in an external procedure written in Java.	255
Maximum size of an ARRAY or VARRAY UDT. This limit does not include the number of bytes used by the row header and the primary index or primary AMP index of a table.	64 KB
Maximum length of external name string for an external routine. An external routine is the portion of a UDF, external procedure, or method that is written in C, C++, or Java (only external procedures can be written in Java). This is the code that defines the semantics for the UDF, procedure, or method.	1,000 characters
Maximum package path length for an external routine.	256 characters
Maximum SQL text size in a procedure.	Approximately 1 MB
Maximum number of nested CALL statements in a procedure.	15
Maximum number of Statement Areas per SQL procedure diagnostics area. See <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148 and <i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.	1
Maximum number of Condition Areas per SQL procedure diagnostics area. See <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148 and <i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.	16

Query and Workload Analysis Limits

Parameter	Value
Maximum number of columns and indexes on which statistics can be collected or recollected at one time. 512 or limited by available parse tree memory and amount of spool.	512
Maximum number of pseudo indexes on which multicolumn statistics can be collected and maintained at one time. A pseudo index is a file structure that allows you to collect statistics on a composite, or multicolumn, column set in the same way you collect statistics on a composite index. This limit is independent of the number of indexes on which statistics can be collected and maintained.	32
Maximum number of sets of multicolumn statistics that can be collected on a table or join index if single-column PARTITION statistics are not collected on the table or index.	32
Maximum number of sets of multicolumn statistics that can be collected on a table or join index if single-column PARTITION statistics are collected on the table or index.	31

Secondary, Hash, and Join Index Limits

Parameter	Value
Number of tables that can be referenced in a join.	128
Minimum number of secondary, hash, and join indexes, in any combination, per base data table.	0
Maximum number of secondary, hash, and join indexes, in any combination, per base data table. Each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in this calculation. The number of system-defined secondary and single-table join indexes contributed by PRIMARY KEY and UNIQUE constraints counts against the combined limit of 32 secondary, hash, and join indexes per base data table.	32
Maximum number of columns referenced per secondary index.	64
Maximum number of columns referenced per single table in a hash or join index.	64
Maximum number of rows per secondary, hash, or join index.	Limited only by disk capacity.
Maximum number of columns referenced in the fixed part of a compressed join index. Vantage implements a variety of different types of user-visible compression in the system. When describing compression of hash and join indexes, compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.	64

Parameter	Value
Maximum number of columns referenced in the repeating part of a compressed join index.	64
Maximum number of columns in an uncompressed join index.	2,048
Maximum number of columns in a compressed join index.	<ul style="list-style-type: none"> • 64 for repeating • 64 for nonrepeating

Reference Index Limits

Parameter	Value
Maximum number of reference indexes per base data table. There is a maximum of 128 Reference Indexes in a table header, 64 from a parent table to child tables and 64 from child tables to a parent table.	64

SQL Request and Response Limits

Parameter	Value
Maximum SQL text size per request. This includes SQL request text, USING data, and parcel overhead.	1 MB
Maximum request message size. The message includes SQL request text, USING data, and parcel overhead.	7 MB
Maximum number of entries in an IN list. There is no fixed limit on the number of entries in an IN list; however, other limits such as the maximum SQL text size, place a request-specific upper bound on this number.	Unlimited
Maximum SQL activity count size.	8 bytes
Maximum number of tables and single-table views that can be joined per query block. This limit is controlled by the MaxJoinTables DBS Control field.	128
Maximum number of partitions for a hash join operation.	50
Maximum number of subquery nesting levels per query.	64
Maximum number of tables or single-table views that can be referenced per subquery. This limit is controlled by the MaxJoinTables DBS Control field..	128
Maximum number of fields in a USING row descriptor.	2,536
Maximum number of bytes in USING data. This value does not include Smart LOB (SLOB) data.	1,040,000
Maximum number of open cursors per embedded SQL program.	16
Maximum SQL text response size.	1 MB
Maximum number of columns per DML request ORDER BY clause.	64

Parameter	Value
Maximum number of columns per DML request GROUP BY clause.	64
Maximum number of fields in a CONSTANT row.	32,768

Row-Level Security Constraint Limits

Parameter	Value
Maximum number of row-level security constraints per table.	5
Maximum number of hierarchical row-level security constraints per user or profile.	6
Maximum number of values per hierarchical row-level security constraint.	10,000
Maximum number of non-hierarchical row-level security constraints per user or profile.	2
Maximum number of values per non-hierarchical row-level security constraint.	256

Session Limits

The session specifications in the following table apply to a single session:

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of sessions per gateway vproc. See <i>Teradata Vantage™ - Database Utilities</i> , B035-1102 for details.	Tunable: 1 - 2,147,483,647. 1,200 maximum certified. The default is 600.
Maximum number of active request result spools per session.	16
Maximum number of parallel steps per request. Parallel steps can be used to process a request submitted within a transaction (which can be either explicit or implicit).	20
Maximum number of materialized global temporary tables that can be materialized simultaneously per session.	2,000
Maximum number of volatile tables that can be instantiated simultaneously per session.	1,000
Maximum number of SQL procedure diagnostic areas that can be active per session.	1

Designing With Task-Oriented Profiles

This section demonstrates how you can use task-oriented views to isolate users from direct contact with the database.

Concepts, Policies, User Profiles, and Rules

Isolating Users from the Database Using Views

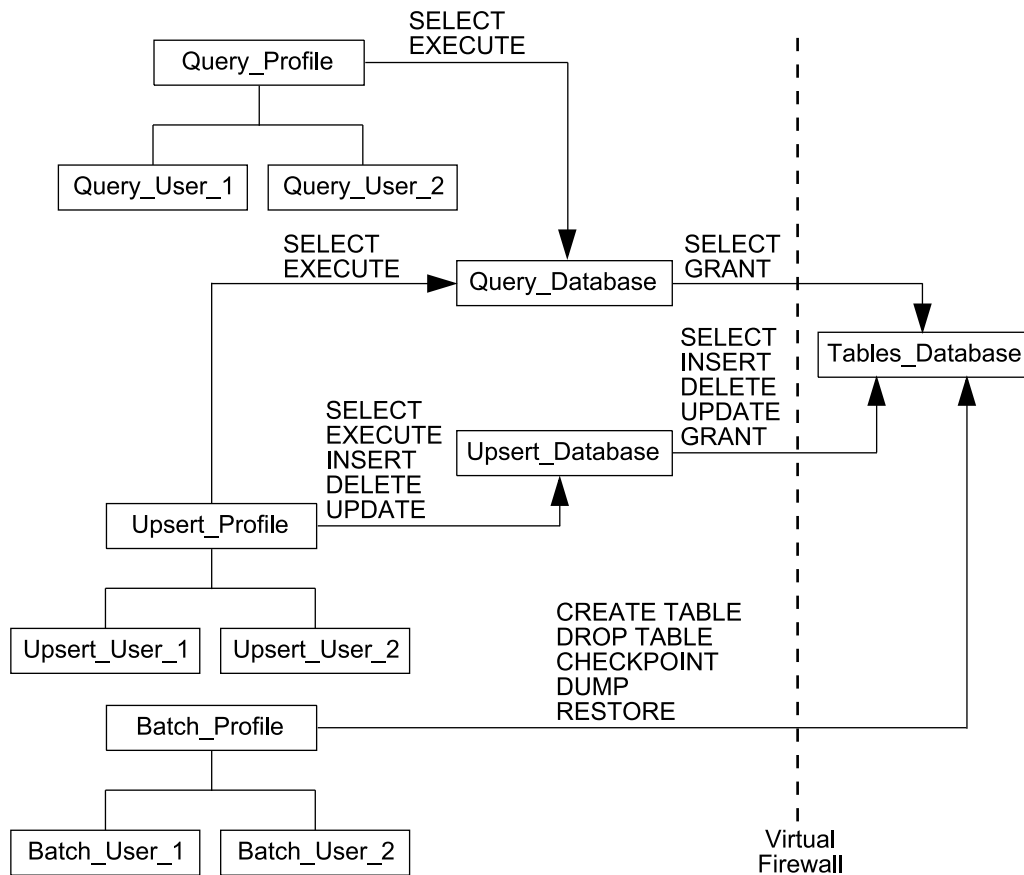
This topic presents a well-designed database structure based on a structure formulated and used by a Teradata customer.

The design incorporates different task-oriented profiles that access only a series of views and macros defined within several task-oriented databases.

Only these task-oriented access definition databases can access the base tables defined for the database. This design structure builds a virtual firewall between all users and the base tables they access.

Flow Diagram of the Database Structure

The following graphic illustrates the work flows and individual task-oriented profiles and database objects defined for this database structure.



Note that Tables_Database contains only base tables and their associated index subtables.

The example database defines three types of end user profile. The following table describes those types.

End User Profiles

Profile	Description	Permitted SQL Statements and Utility Commands
Query_Profile	Defines privileges to the views, macros, and stored procedures defined in Query_Database.	<ul style="list-style-type: none"> EXECUTE SELECT
Upsert_Profile	Defines privileges to the views, macros, and stored procedures defined in Upsert_Database.	<ul style="list-style-type: none"> DELETE EXECUTE INSERT SELECT UPDATE
Batch_Profile	Defines privileges to the tables defined in Tables_Database.	<ul style="list-style-type: none"> CHECKPOINT CREATE TABLE DROP TABLE DUMP

Profile	Description	Permitted SQL Statements and Utility Commands
		<ul style="list-style-type: none"> RESTORE

Rules for End User Profiles

The following rules apply to all user profiles except Batch_Profile:

- All users belong to one or more task profiles.
- Users inherit their privileges from the task profiles to which they belong.
- A user can belong to more than one task profile.
- Privileges are granted at DATABASE or USER levels only.
- Query_Profile and Upsert_Profile only have access to databases that contain macros, stored procedure definitions, and views exclusively.
- Batch_Profile is the only profile that permits the base tables in Tables_Database to be accessed directly.

Rules for Batch_Profile

- Users inherit their privileges from Batch_Profile.
- Privileges are granted at Database or User levels only.
- Batch_Profile has direct access to the base tables in Tables_Database because of the functions performed using it.
- Batch_Profile users can perform the following functions against Tables_Database
 - CHECKPOINT transactions
 - CREATE TABLEs
 - DROP TABLEs
 - DUMP databases
 - RESTORE databases

Rules for Query_Database

Note that Query_Database objects are organized in such a way that the entire database can be archived in one operation. You can also archive Upsert_Database in the same operation if you choose to do so.

The following rules apply to Query_Database.

- Both Query_Profile and Upsert_Profile users can perform the following functions against Query_Database.
 - CALL permitted stored procedures
 - EXECUTE permitted macros
 - SELECT rows from Tables_Database objects through views

- The database contains only views, stored procedure definitions, and macros that permit an end user to query or grant privileges on Tables_Database objects indirectly.

Rules for Upsert_Database

Note that Upsert_Database objects are organized in such a way that the entire database can be archived in one operation. You can also archive Query_Database in the same operation if you choose to do so.

The word upsert derives from update-insert and implicitly refers to delete as well.

Only Upsert_Profile users exclusively can perform the following functions against Upsert_Database:

- CALL permitted stored procedures
- DELETE rows in Tables_Database objects through views
- EXECUTE permitted macros
- INSERT rows into Tables_Database objects through views
- SELECT rows from Tables_Database objects through views
- UPDATE rows in Tables_Database objects through views

The database contains only views, stored procedure definitions, and macros that permit an end user to perform the following actions on Tables_Database objects indirectly.

- DELETE rows
- INSERT rows
- SELECT rows
- UPDATE rows
- GRANT privileges on database objects

Summary Physical Design Scenario

This section presents an overview of the physical design process.

The following scenario assumes that you are pursuing a 3NF logical model for your database schema.

Prerequisites for the Process Review

Use the following checklist to ensure you are ready to enter the physical design phase of your project.

√	Physical Design Phase Entry Criteria
	Fully documented, fully normalized logical model
	Fully documented column and table demographics
	Good understanding of Teradata architecture-specific physical design principles

Process Review

The following are the high-level stages in the physical design process:

1. Identify all index and partitioning candidates.
2. Review table demographics, then verify or modify index and partitioning choices.
3. Evaluate denormalization possibilities, including dimensional views, then verify or modify index and partitioning choices.
4. Evaluate derived data, then verify or modify index and partitioning choices.
5. Implement the system and initiate production processing.
6. Track changing demographics.
7. Review indexes and partitioning periodically to ensure they are still the best choices.
8. Perform appropriate EXPLAIN request modifiers periodically to track whether defined indexes are being selected for query processing by the Optimizer and whether the current partitioning is still facilitating row and column partition elimination.
9. Reevaluate table use.
10. Document any changes.
11. Archive older versions of your findings for use in analyzing trends.

Sample Worksheet Forms

This section contains a complete set of forms that you can print or photocopy and use for your database design projects.

The following forms are included:

- Domains Form
- Constraints Form
- System Form
- Report/Query Analysis Form
- Table Access Summary By Column Form
- Table Form
- Row Size Calculation Form for Byte-Packed Format Systems, parts 1 and 2
- Row Size Calculation Form for Byte-Aligned Format Systems, parts 1 and 2

Because several data types have different storage sizes on byte-packed format and 64-bit byte-aligned format systems (see [Database-Level Capacity Planning Considerations](#) for details), separate forms are provided for the two architectures to minimize the possibility of mistakenly entering column values that are either too small or too large.

[illegible]

Data Types

Bigint	I8	Interval Month	IMO	PERIOD(Time)	PT
BLOB	BL(n)	Interval Day	ID	PERIOD(Time With Time Zone)	PTTZ
Byte	B(n)	Interval Day To Hours	IDH	PERIOD(Timestamp)	PTS
Byteint	I1	Interval Day To Minute	IDM	PERIOD(Timestamp <UNTIL_CHANGED>)	PTS_UC
Character	C(n)	Interval Day To Second	IDS	PERIOD(Timestamp With Time Zone)	PTSTZ
Character Varying	CV(n)	Interval Hours	IH	PERIOD(Timestamp With Time Zone <UNTIL_CHANGED>)	PTSTZ_UC
CLOB	CL(n)	Interval Hour To Minute	IHM	Real	R
Date	D	Interval Hour To Second	IHS	Smallint	I2
Decimal	DEC(n,m)	Interval Minute	IM	Time	T
Double Precision	DP	Interval Minute To Second	IMS	Timestamp	TS
Float	F(n)	Interval Second	IS	Time With TimeZone	TTZ
Graphic	G(n)	Long Character Varying	LVC	Timestamp With TimeZone	TSTZ
Integer	I	Long Graphic Varying	LVG	Varbyte	VB
Interval Year	IY	Numeric	NUM(n,m)	Varchar	VC(n)
Interval Year to Month	IYM	PERIOD(Date)	PD	Vargraphic	VG(n)

Constraints

Page: _____ Of _____

ELDM Page: _____

System: _____

[illegible]

System

Page: _____ Of _____

ELDM Page: _____

System: _____

System Name: _____

System Description: _____

[illegible]

Report/Query Analysis Form

Frequency or Importance Ranking: _____

Report/Query Description: _____

Table:

Number Of Output Rows:

Input Value Or Source	
Access Column(s)	

Table:

Number Of Output Rows:

Input Value Or Source	
Access Column(s)	

Table:

Number Of Output Rows:

Input Value Or Source	
Access Column(s)	

Table:

Number Of Output Rows:

Input Value Or Source	
Access Column(s)	

Table Access Summary By Column

Page: _____ Of _____

ELDM Page: _____

System: _____

Table Name: _____

Column Names: _____

Value Accesses			Join Accesses		
Transaction ID	Rows	Freq	Transaction ID	Rows	Freq
Value Accesses Totals:			Join Accesses Totals:		

Table		ELDM Page: _____	
Page: _____ Of _____		System: _____	
Table Name	Table Type	Cardinality	Data Protection
Column Name			
PK/FK/ID			
Constraint Number			
Value Access Frequency			
Joint Access Frequency			
Joint Access Rows			
Distinct Values			
Maximum Rows/Value			
Maximum Rows Null			
Typical Rows/Value			
Change Rating			
PI/PA/NoPI/SI			
Sample Data			

Row Size Calculation Form for Packed64 Systems - Page 1 of 2

Table Name:

Variable Data Detail

[illegible]

Sum(a)=SUM of the AVERAGE number of bytes expected for the variable column.

Sum(n)=SUM of the CHAR and GRAPHIC column bytes.

**Round up to an even number of bytes

Row Size Calculation Form for Packed64 Systems - Page 2 of 2

Table Name:

Variable Data Detail

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

Row Size Calculation Form for Aligned Row Format Systems - Page 1 of 2

Table Name:

Variable Data Detail

[illegible]

Sum(a)=SUM of the AVERAGE number of bytes expected for the variable column.

Sum(n)=SUM of the CHAR and GRAPHIC column bytes.

**Round up to an even number of bytes

Row Size Calculation Form for Aligned Row Format Systems - Page 2 of 2

Table Name:

Variable Data Detail

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

UDT Name	Number of Columns	Sizing Factor	Total
		* 1 =	
		* 2 =	
		* 4 =	
		* 4 =	
		* 6 =	
		* 8 =	
		* 10 =	
		* 12 =	
		* 2 =	
		* 4 =	
		* 2 =	
		* 2 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 10 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 2 =	
		* 6 =	
		* 6 =	
		* 1 =	
		* 2 =	
		* 4 =	
		* 8 =	
		* 39 =	
		* 8 =	
		SUM(n) =	

Designing Tables for Optimal Performance

You can optimize some of the CREATE TABLE and ALTER TABLE table options to improve efficiency in accessing and maintaining the table.

Minimizing Table Size

Request performance is directly proportional to the size of the set of tables accessed. As table size increases, Vantage requires additional I/O operations to retrieve or update data.

For large tables that cannot be reduced in size, you can use the secondary index, single-table join index, primary index, primary AMP index, and partitioning features to limit the portion of a table that must be scanned to access needed data.

Reducing the Number of Table Columns

As the number and size of columns increases, the row size necessarily increases. A row cannot occupy more than one data block, so if a row exceeds the specified maximum data block size, the system returns an error message.

Adjusting the DATABLOCKSIZE and MERGEBLOCKRATIO Table Parameters

You can control the default size for multirow data blocks on a table-by-table basis in a CREATE TABLE or ALTER TABLE request using the DATABLOCKSIZE and MERGEBLOCKRATIO options:

Disk arrays can scan at higher rates if the I/Os are larger, but larger I/Os can be less efficient for row-at-a-time access which requires that the entire data block be read for the relatively few bytes contained in a row. Cylinder reads enable smaller data blocks for row-at-a-time access and large reads for scans.

For average workloads, the benefits of using large data blocks outweighs the small penalty associated with row-at-a-time access up to 64 KB. Setting the data block size requires more judgment and analysis for 128KB and 1MB data blocks where the penalty for row-at-a-time access becomes measurable.

IF you specify...	THEN...
DATABLOCKSIZE in CREATE TABLE	<p>the data block can grow to the size specified in DATABLOCKSIZE instead of being limited to the global PermDBSize (see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102).</p> <p>For any row, Vantage uses only the data block size required to contain the row.</p> <p>Note:</p> <p>If you use block level compression, adjust DATABLOCKSIZE to the maximum, 255 sectors.</p>

IF you specify...	THEN...
DATABLOCKSIZE in ALTER TABLE	the data blocks can grow to the size specified in DATABLOCKSIZE when the row size requires the growth. Whether data blocks are adjusted to that new size immediately or gradually over a long period of time depends on the use of the IMMEDIATE clause.
MERGEBLOCKRATIO in CREATE TABLE	Vantage limits attempts to combine blocks if the result is larger than the specified percent of the maximum multirow data block size.
MERGEBLOCKRATIO in ALTER TABLE	the size of the resulting block when multiple existing data blocks are being merged has a upper limit. The limit depends on whether Vantage determines that logically adjacent data blocks can be merged with the single data block being modified or not. Data blocks can still be initially loaded at the PermBDSIZE specification or the data block size specified with the DATABLOCKSIZE option. Merges occur only during full-table modifications.
the IMMEDIATE clause	the rows in all existing data blocks of the table are repacked into data blocks using the newly specified size. For large tables, this can be a time-consuming operation, requiring spool space to accommodate 2 copies of the table while it is being rebuilt. If you do not specify the IMMEDIATE clause, existing data blocks are not modified. When individual data blocks of the table are modified as a result of user transactions, the new value of DATABLOCKSIZE is used. Thus, the table changes over time to reflect the new data block size.

To specify the global data block size, use PermBDSIZE (see *Teradata Vantage™ - Database Utilities*, B035-1102).

Adjusting FREESPACE

You can specify the default value for free space left on a cylinder during certain operations on a table-by-table basis by specifying the FREESPACE parameter in the CREATE TABLE and ALTER TABLE requests.

This option enables you to select a different value for tables that are constantly modified versus tables that are only read after they are loaded. To specify the global free space value, use the FreeSpacePercent command of the DBS Control utility (see *Teradata Vantage™ - Database Utilities*, B035-1102).

Using Identity Columns, Compression, and Referential Integrity for Optimal Performance Design

The following sections in this document contain information about designing tables optimally using referential integrity and compression, respectively.

- [Primary Index, Primary AMP Index, and NoPI Objects](#)
- [Designing for Database Integrity](#)
- [Database-Level Capacity Planning Considerations](#)

Using Indexes to Enhance Performance

The following sections in this document contain information about designing tables optimally using indexes.

- [Indexes and Maps](#)
- [Primary Index, Primary AMP Index, and NoPI Objects](#)
- [Secondary Indexes](#)
- [Join and Hash Indexes](#)

Understanding the Effects of Altering Tables

Using the ALTER TABLE statement can affect system performance and space requirements.

Note:

Changes to the Data Dictionary resulting from these actions have minimal effect on performance.

Action	Performance Impact	Space Requirements
Add a column (COMPRESS, NULL)	All table rows are changed if a new presence byte is added.	Slight increase in required permanent space.
Add a column (NOT NULL, DEFAULT, and WITH DEFAULT)	<ul style="list-style-type: none"> • All table rows change (without column partitioning). • Only affected column partitions change (with column partitioning). 	Increase in required permanent space.
Add a column (NULL, fixed-length)	All table rows are changed.	Increase in required permanent space.
Add a column (NULL, variable length)	All table rows are changed.	Slight increase in required permanent space.
Add FALLBACK	Entire table is accessed to create the fallback copy. Long-term performance effects.	Approximately doubled the required permanent space.
Add CHECK constraints	Takes time to validate rows, which impacts performance.	Unchanged.
Add referential integrity	Takes time to check data. Impacts performance long term. Similar to adding indexes.	Possible large increase in the following. <ul style="list-style-type: none"> • Spool space. • Permanent space (for index if not soft batch).
Change the format, title, default	No impact.	Unchanged.
Change the cylinder free space percent	<ul style="list-style-type: none"> • Raising the free space percent can make inserting new data less 	Increase in required permanent space for operations

Action	Performance Impact	Space Requirements
	<p>expensive by reducing migration and cylinder allocations.</p> <ul style="list-style-type: none"> Lowering the free space percent has the opposite effect. 	such as default maximum, MultiLoad, restore.
Change the maximum multirow block size	<ul style="list-style-type: none"> If the table is not updated after the change, then there is no impact. If the table is changed, there can be performance impact whether or not the IMMEDIATE clause is specified. 	<ul style="list-style-type: none"> Slight increase in required permanent space for smaller values. Slight decrease in required permanent space for larger values.
Delete the FALLBACK option	<p>FALLBACK subtable is deleted. Long-term performance effects.</p> <p>Note: You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.</p>	Approximately half the required permanent space.
Drop a column	All table rows are changed.	Decrease in required permanent space.

Compression Methods

This section describes the compression methods available in Vantage.

The following information is presented for each compression method:

- Benefits
- Best application
- Usage restrictions and limitations

Multivalue Compression

This section summarizes the benefits, best uses, and restrictions of multivalue compression in Vantage.

Benefits of Multivalue Compression

- Minimal performance decrement for compressing or decompressing values
- Reduced I/O for table scans, spooling result sets, and data loads
- Reduced I/O for cached tables because more smaller tables can be cached, eliminating I/O operations on those tables
- Valid for most commonly used data types
- Spools retain the multivalue compression of their base tables
- Can be specified for permanent, global temporary, and volatile table columns
- Automatically compresses nulls
- Easily and complementarily combined with algorithmic compression
- Easily combined with block-level and temperature-based block-level compression and autocompression for column-partitioned tables and join indexes

Best Use of Multivalue Compression

- Very wide columns
- Columns with few distinct values
- Columns in large tables
- Tables having a large number of rows with repeating values
- Tables having a large number of rows with default values
- Columns having a constant distribution of values across time

Restrictions and Limitations of Multivalue Compression

- Requires extensive data analysis prior to implementation
- Can compress a maximum of only 255 different characters per column

- The more values compressed per row, the more presence bits in the row header are required, which increases the row size
- Affects table header size
- Cannot specify for primary index or primary AMP index columns
- Cannot for the partitioning columns of a partitioning expression
- Can specify on secondary index columns, but the column data in the secondary index subtable is not multivalue compressed
- Columns with many unique or nearly unique values cannot benefit from multivalue compression
- A given multivalue compression scheme can become less useful across time if the values in the table change significantly

Algorithmic Compression

This section summarizes the benefits, best uses, and restrictions of algorithmic compression in Vantage.

Benefits of Algorithmic Compression

- Easy to define
- Can be specified for permanent, global temporary, and volatile table columns
- Compresses compressible columns in all table rows regardless of their value

When combined with multivalue compression for the same column, algorithmic compression only compresses the values that are not compressed by multivalue compression

- Has no effect on table header size
- Spools retain the algorithmic compression of their base tables
- No limit to the number of table columns that can be compressed using algorithmic compression
- No limit on the size of algorithmically compressed values
- Useful for columns with many unique or nearly unique values
- Easily and complementarily combined with multivalue compression
- Easily combined with autocompression for column-partitioned tables and join indexes
- Can specify algorithmic compression on a secondary index column, though the column data in the secondary index subtable is not algorithmically compressed
- Can specify algorithmic compression on permanent and global temporary table columns
- Automatically compresses nulls

Best Use of Algorithmic Compression

- Very wide character columns, particularly Unicode columns
- Columns that contain click stream data

Restrictions and Limitations of Algorithmic Compression

- Both compression and decompression consume CPU resources, causing negative performance issues

Performance issues are restricted to when an algorithmically compressed column is referenced directly

- If you use your own compression algorithms, you must write and test your own compression UDFs to implement them
- Can be used for only a few data types
- Compressed values are stored within the row
- Best not used in combination with block-level compression because of the combined cost of their decompression
- Cannot specify for primary index or primary AMP index columns
- Cannot specify for the partitioning columns of a partitioning expression

Block-Level Compression

This section summarizes the benefits, best uses, and restrictions of block-level compression in Vantage.

Benefits of Block-Level Compression

- Can apply to data blocks that contain nearly any type of data
- Requires no analysis of column data prior to implementation
- Easily combined with multivalue compression
- More efficient than algorithmic compression for tables with numerous narrow-width columns
- Has no effect on table header size

Best Use of Block-Level Compression

- Very large tables
- Tables that do not require frequent decompression, especially tables that contain rarely updated COLD or static data
- Tables accessed with low concurrency
- Systems with low CPU utilization levels
- Systems where you can use workload management to restrict the overhead of data decompression to a minimal level

Restrictions and Limitations of Block-Level Compression

- Strictly used to reduce storage space
- Does not have any performance benefits
- Can be very costly due to the following factors:
 - Initial cost of applying the compression
 - Ongoing cost of decompressing data blocks when they are accessed
 - Ongoing cost of recompressing data blocks after they have been modified
- Should not apply for tables that contain WARM or HOT data if they have a CPU utilization rate greater than 80%

- Extent of compression that can be achieved depends on the characteristics of the data contained within a data block as well as the maximum size that has been specified for permanent data blocks

Individual data blocks within a table might have different size reduction rates, so data blocks frequently show more size variance after being block-level compressed than they did in their original state

- Best not used in combination with algorithmic compression because of the combined cost of their decompression
- Cannot use for data dictionary tables, WAL data blocks, cylinder indexes, table headers, or other internal file structures
- If either multivalue compression or algorithmic compression is defined on a table, the extent of storage reduction achieved by adding block-level compression is less than if no compression had been present at the time block-level compression was added
- Achieves the best compression ratio for data blocks defined with the maximum 1 MB size

Larger maximum data block sizes at the time of compression allow cylinders to be more fully utilized, which can also impact the degree of compressibility.

As a table matures, it experiences updates and data block splits, which reduce the size of some data blocks.

During the compression process, Vantage reforms data blocks into their maximum size.

With larger block sizes defined, Vantage can achieve a greater degree of block-level compression.

Note:

For performance purposes, the size to make data blocks depends on the application (as it does without BLC). Random access (for example, prime key access and USI access paths) does better with more moderate data block sizes, but sequential workloads (full table scans) work better with larger data blocks.

- A large number of significantly smaller data blocks can fill the cylinder index, leaving a cylinder with space that cannot be used because some data blocks are considered too small to provide meaningful compression, and Vantage bypasses those blocks during the compression process
- Does not compress secondary index subtables, so if a table has secondary indexes before compression, the size reduction rate after compression appears to be smaller than if the table had no secondary indexes at the time of compression
- Should not use for small or medium-sized tables that are frequently accessed or updated.

Temperature-Based Block-Level Compression

This section summarizes the benefits, best uses, and restrictions of temperature-based block-level compression in Vantage.

Benefits of Temperature-Based Block-Level Compression

- Controlled by Teradata Virtual Storage

- Can specify for all user data tables, or can apply it at the level of individual tables

When applied to an individual table, impacts all compressible subtables and structures associated with that table.

When applied at the level of individual tables, specified compression is inherited by all of the compressible subtables associated with that table, including the following:

- Primary data tables
- Fallback data tables
- Index fallback subtables
- CLOB subtables
- Can use query bands in addition to automatic compression to manage the compression status of tables under the control of temperature-based block-level compression
- Can use query bands to load temperature-sensitive tables in either compressed or uncompressed format if you want a uniform compression state and you know the expected temperature of the data being loaded

For more information about using query bands with temperature-based block-level compression, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

- Has no effect on table header size

Best Use of Temperature-Based Block-Level Compression

- Large tables where a WARM or HOT subset of their data is accessed frequently
- Tables whose temperature profile is relatively stable across time
- Large row-partitioned tables whose activity is concentrated in currently WARM or HOT row partitions, but whose older row partitions are relatively dormant and COLD
- Column-partitioned tables because their frequently accessed data is grouped together in columnar partitions that are WARM or HOT and other column partitions that are relatively dormant and COLD

Restrictions and Limitations of Temperature-Based Block-Level Compression

- Only applies to cylinders containing permanent tables
- Can make the data have a warmer temperature than it had when it was uncompressed because after compression and cylinder consolidation, each cylinder contains more data than it previously had
- Because Teradata Virtual Storage measures temperature at the cylinder level, a greater number of data blocks could mean a higher cylinder access count

Row Compression

This section summarizes the benefits, best uses, and restrictions of row compression in Vantage.

Benefits of Row Compression

- Reduces storage space by storing a repeating column value set only once, while any non-repeating column values that belong to that set are stored as logical extensions of the base repeating set
- Assigned by default for hash indexes
- Like multivalue compression, no decompression is necessary to access row-compressed data values
- Has no effect on table header size

Best Use of Row Compression

- Hash and join indexes that have numerous column values that repeat across rows.

Restrictions and Limitations of Row Compression

- Only applies to hash and join indexes
- Must be explicitly specified in the CREATE JOIN INDEX request for a join index at the time the index is created

For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Autocompression

This section summarizes the benefits, best uses, and restrictions of autocompression in Vantage.

Benefits of Autocompression

- Default for column partitions. If you change the default to NO AUTO COMPRESS or you specify NO AUTO COMPRESS for the column partitioning, you can specify AUTO COMPRESS for a column partition to have that column partition autocompressed.
- Vantage can choose to apply multiple autocompression methods from a list of available techniques depending on their effectiveness for a given column partition (note that currently there are no available techniques for column partitions with ROW format).
- Reduces I/O overhead if the data can be compressed.
- Vantage only needs to decompress autocompressed column partition values that are accessed, and decompression is automatically done with typically very little performance overhead when they are retrieved.
- Easily combined with null, multivalue, algorithmic, row, block-level, and temperature-based block-level compression.
- Has no effect on table header size.

Best Use of Autocompression

- Column partitions with a single column and COLUMN format

Restrictions and Limitations of Autocompression

- There is CPU overhead in determining whether or not a physical row should be compressed and in determining what compression techniques to use if it is to be compressed.
This overhead is usually minimal, but you can eliminate it by specifying the NO AUTO COMPRESS option for a column partition or the column partitioning (or change the default to NO AUTO COMPRESS), which eliminates autocompression for that column or column partition.
- There is additional CPU overhead when you insert rows into a column-partitioned table that uses autocompression.
- If you specify autocompression, Vantage might not apply user-specified MVC or algorithmic compression if they do not further reduce storage space.
- There are no autocompression techniques applied to a column partition with ROW format, even when AUTO COMPRESS is the default for the column partition or you specify it directly.

Row Header Compression

This section summarizes the benefits, best uses, and restrictions of row header compression in Vantage.

Benefits of Row Header Compression

- Only applied to columns partition that have COLUMN format
- Reduces I/O overhead
- Automatically applied to column partitions with COLUMN format whether you specify AUTO COMPRESS or NO AUTO COMPRESS

Best Use of Row Header Compression

- Column partitions in COLUMN format that contain many column partition values per container

Restrictions and Limitations of Row Header Compression

- If only a few column partition values can be placed in a container because of their width, there can be an increase in the space needed for a column-partitioned object in COLUMN format compared to the object without column partitioning

In this case, ROW format might be a more appropriate choice

- If a combined partition contains only a few column partition values, there can be a very large increase in the space needed for a column-partitioned object compared to the same object defined without column partitioning

In this case, consider either altering the row partitioning to allow for more column partition values per combined partition, reducing the number of row partitions, or removing column partitioning altogether

- Over partitioning a table loses the advantages of row header compression

References

Teradata Publications Related to Database Design

Jerry Klindt, Paul Sinclair, Steve Molini, Jeremy Davis, Rama Krishna Korlapati, Hong Gui, and Stephen Brobst, *Partitioned Primary Index Usage Orange Book*, 541-0003869.

No Primary Index (NoPI) Table User Guide Orange Book, 541-0007565.

Increased Partition Limit and Other Partitioning Enhancements Orange Book, 541-0009027.

Teradata® Columnar Primer Orange Book, TDN0009884

Colin White, "In the Beginning: An RDBMS History," *Teradata Magazine*, 4(3):32-39, 2004.

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community