

## DB2 9 DBA exam 731 prep, Part 2: Data placement

Dwaine Snow

05 July 2006

Learn how to create DB2 databases, and about the mechanisms available for storing your tables, indexes, and data within the database. This tutorial focuses on partitioning, compression, and XML, which are all important performance and application development concepts that you need to know to store and access data quickly and efficiently. This is the second in a [series of seven DB2 DBA certification prep tutorials](#) that you can use to help prepare for the DB2 V9 for Linux, UNIX, and Windows Database Administration (Exam 731).

[View more content in this series](#)

### Before you start

#### About this series

If you are preparing to take the DB2 DBA certification exam 731, you've come to the right place -- a study hall, of sorts. This [series of seven DB2 certification preparation tutorials](#) covers the major concepts you'll need to know for the test. Do your homework here and ease the stress on test day.

#### About this tutorial

This tutorial discusses the creation of DB2 databases, as well as the various methods used for placing and storing objects within a database. This tutorial focuses on partitioning, compression, and XML, which are all important performance and application development concepts that you need to know to store and access data quickly and efficiently. This tutorial is the second in a series of seven tutorials that you can use to help you prepare for the DB2 Database Administration Certification (Exam 731). The material in this tutorial primarily covers the objectives in Section 2 of the exam, titled Data Placement. You can view these objectives at: <http://www-03.ibm.com/certify/tests/obj731.shtml>.

You should also review the [Resources](#) at the end of this tutorial for more information about DB2 server management.

### Objectives

After completing this tutorial, you should be able to:

- Create databases
- Use schemas
- Determine the various table space states
- Create and manipulate DB2 objects
- Create an SMS table space and understand its characteristics
- Understand the characteristics of, and use, DB2's automatic storage
- Implement table partitioning and MDC on your tables
- Use table compression
- Use XML

## Prerequisites

To understand some of the material presented in this tutorial, you should be familiar with the following terms:

- **Object:** Anything in a database that can be created or manipulated with SQL (for example, tables, views, indexes, packages).
- **Table:** A logical structure that is used to present data as a collection of unordered rows with a fixed number of columns. Each column contains a set of values, each value of the same data type (or a subtype of the column's data type); the definitions of the columns make up the table structure, and the rows contain the actual table data.
- **Record:** The storage representation of a row in a table.
- **Field:** The storage representation of a column in a table.
- **Value:** A specific data item that can be found at each intersection of a row and column in a database table.
- **Structured Query Language (SQL):** A standardized language used to define objects and manipulate data in a relational database. (For more on SQL, see the fourth tutorial in this series.
- **DB2 optimizer:** A component of the SQL precompiler that chooses an access plan for a Data Manipulation Language (DML) SQL statement by modeling the execution cost of several alternative access plans and choosing the one with the minimal estimated cost.

To take the DB2 9 DBA exam, you must have already passed the [DB2 9 Fundamentals exam 730](#). We recommend that you take the [DB2 Fundamentals tutorial series](#) before starting this series.

Although not all materials discussed in the Family Fundamentals tutorial series are required to understand the concepts described in this tutorial, you should at least have a basic knowledge of:

- DB2 products
- DB2 tools
- DB2 instances
- Databases
- Database objects

## System requirements

You do not need a copy of DB2 to complete this tutorial. However, you will get more out of the tutorial if you download the free trial version of [IBM DB2 9](#) to work along with this tutorial.

## Creating a database

### Database directories

A *system database directory* file exists for each instance of the database manager, and contains one entry for each database that has been cataloged for this instance. Databases are implicitly cataloged when the `create database` command is issued, and can also be explicitly cataloged with the `catalog database` command.

A *local database directory* file exists in each drive or path in which a database has been defined. This directory contains one entry for each database accessible from that location.

### Creating a database

When you create a database, each of the following tasks are done for you:

- Setting up of all the system catalog tables that are needed by the database
- Allocation of the database recovery log
- Creation of the database configuration file and the default values set
- Binding of the database utilities to the database

### The `create database` command

To create a database, use the command:

```
create database
```

You can optionally specify the following:

- Storage paths
- Database partition number for the catalog partition
- Drive or path on which to create the database
- Codeset and territory
- Collating sequence
- Default extent size
- Whether the database should be automatically configured
- Table space definitions for the CATALOG, TEMPORARY, and USERSPACE1 table spaces

### The default database

The `create database` command creates *three* default table spaces:

#### SYSCATSPACE

For the system catalog tables. SYSCATSPACE cannot be dropped.

#### TEMPSPACE1

For system-created temporary tables. The TEMPSPACE1 table space can be dropped once another temporary table space has been created.

#### USERSPACE1

The default table space for user-created objects. The USERSPACE1 table space can be dropped once another user-created table space has been created.

## The system catalogs

A set of system catalog tables is created and maintained for each database. These tables contain information about the definitions of the database objects (tables, views, indexes, and packages, for example) and security information about the type of access that users have to these objects. These tables are stored in the SYSCATSPACE table space.

## The directory structure

The `create database` command lets you specify the drive or directory on which to create the database, depending on the operating system.

If no drive or directory is specified, the database is created on the path specified by the DFTDBPATH instance (database manager) configuration parameter.

If no drive or directory is specified, *and* the DFTDBPATH instance level configuration parameter is not set, the database is created on the drive or path where the `create database` command was executed.

The `create database` command creates a series of subdirectories. The first subdirectory is named after the instance owner for the instance in which the database was created. Under this subdirectory, DB2 creates a directory that indicates which database partition the database was created in.

For a non-partitioned database the directory will be NODE0000. For a partitioned database, the directory will be named NODExxxx, where xxxx will be the four digit partition number for the database instance as designated in the db2nodes.cfg file. For example, for partition number 43, the directory would be NODE0043.

In Windows, instances do not really have an instance owner, so the name of the instance (for example, DB2) will be used in place of the instance owner's ID.

Because more than one database can be created on the same drive or directory, each database must have its own unique subdirectory. Under the NODExxxx directory, there will be an SQLxxxxx directory for every database that was created on the drive or directory. For example, imagine that we have two databases, DBASM and SAMPLE, that were both created on the C: drive on Windows. There will be two directories: SQL00001 and SQL00002.

To determine the directory under which the database was created, enter the command `list database directory on c:`. This will produce output similar to the following:

```

C:\>db2 list db directory on c:

Local Database Directory on c:

Number of entries in the directory = 2

Database 1 entry:

Database alias           = DBASM
Database name            = DBASM
Database directory       = SQL00002
Database release level   = b.00
Comment                  =
Directory entry type     = Home
Catalog database partition number = 0
Database partition number = 0

Database 2 entry:

Database alias           = SAMPLE
Database name            = SAMPLE
Database directory       = SQL00001
Database release level   = b.00
Comment                  = A sample database
Directory entry type     = Home
Catalog database partition number = 0
Database partition number = 0

C:\>

```

In the example above, the database SAMPLE was created in the SQL00001 directory and the database DBASM was created in the SQL00002 directory under the NODExxxx directory.

By default:

- The system catalog table space (SYSCATSPACE) will use the directory SQLT0000.0
- The system temporary table space (TEMPSPACE1) will use the directory SQLT0001.0
- The default user table space (USERSPACE1) will use the directory SQLT0002.0

## Example Linux/UNIX create database commands

To create a database on the directory (file system) /database, use the following command:

```
create database sample on /database
```

If this command were executed in the instance named dbinst, on the server where database partition 0 is defined, the following directory structures would be created:

- /database/dbinst/NODE0000/sqlldbdir
- /database/dbinst/NODE0000/SQL00001

## Example Windows create database commands

To create a database on the D: drive, use the following command:

```
create database sample on D:
```

If this command were executed in the instance named dbinst, on the server where database partition 0 is defined, the following directory structures would be created:

- D:\dbinst\NODE0000\sqlldbdir
- D:\dbinst\NODE0000\SQL00001

## Creating the USERSPACE1 table space as DMS

To create a database and define the USERSPACE1 table space to be database managed space (DMS), using two file containers, use the following commands:

On Linux or UNIX:

```
create database sample2 user table space managed by database
    using(file '/dbfiles/cont0' 5000, file '/dbfiles/cont1' 5000)
```

On Windows:

```
create database sample2 user table space managed by database
    using(file 'c:\dbfiles\cont0' 5000, file 'c:\dbfiles\cont1' 5000)
```

## Creating the TEMPSPACE1 table space with user-defined containers

To create a database and define the TEMPSPACE1 table space to use two SMS containers (see [SMS table spaces](#)), use the following commands:

On Linux or UNIX:

```
create database sample3 temporary tablespace managed by system
    using('/dbfiles/cont0', '/dbfiles/cont1')
```

On Windows:

```
create database sample3 temporary tablespace managed by system
    using('c:\dbfiles\cont0', 'c:\dbfiles\cont1')
```

## Changing the collating sequence for the database

The command (in Linux and UNIX):

```
create database SAMPLE on /mydbs collate using identity
```

or on Windows:

```
create database SAMPLE on D: collate using identity
```

creates a database and compares strings byte for byte, since the collating sequence has been set to `identity`.

## Automatic storage

### What is a automatic storage?

*Automatic storage*, which is new in DB2 V9, allows you to specify one or more storage paths for a database. Then when you create table spaces, they are automatically placed on the storage paths by DB2. You can enable or configure automatic storage for a database when it is created, as follows:

```
db2 create database db_name automatic storage yes
db2 create database db_name on db_path1, db_path2
```

You can add additional storage paths to a database set up for automatic storage using the `add storage` parameter, as follows:

```
db2 alter database db_name add storage on db_path3
```

## Using automatic storage

Once your database has been set up for automatic storage, you can create table spaces using this mechanism. You have several ways to take advantage of automatic storage once the database has been set up that way. You can simply create a table space in the database (once you are connected to the database), as follows:

```
db2 create tablespace ts_name
```

Or, you can create a table space and specify its initial size and growth characteristics, as follows:

```
db2 create tablespace ts_name
  initialsize 10M
  increasesize 10M
  maxsize 100M
```

In this example the table space will start out at 10MB, and as it gets close to being full, DB2 will automatically extend it by 10MB at a time, up to its maximum size of 100MB.

If the database was not set up for automatic storage, you can still use automatic storage for a table space if you create it and specify its storage:

```
db2 create tablespace ts_name
  managed by automatic storage
```

## Using schemas

### What is a schema?

A *schema* is a high-level qualifier for database objects created within a database. It is a collection of database objects such as tables, views, indexes, or triggers. It provides a logical classification of database objects.

While you are organizing your data into tables, it may also be beneficial to group tables and other related objects together. This is done by defining a schema using the `create schema` command. Information about the schema is kept in the system catalog tables of the database to which you are connected. As other objects are created, they can be placed within this schema.

### System schemas

A set of system schemas are created with every database and placed into the SYSCATSPACE table space:

**SYSIBM**

The base system catalogs. Direct access is not recommended.

**SYSCAT**

SELECT authority granted to PUBLIC on this schema. Catalog read-only views.

Recommended way to obtain catalog information.

**SYSSTAT**

Updatable catalog views -- influences the optimizer.

**SYSFUN**

User-defined functions.

**How is a schema used in DB2?**

Use a schema to fully qualify a table or other object name, as follows:

```
schemaname.tablename
```

You can have multiple tables with the same name but different schema names. Thus, the table `user1.staff` is not the same as `user2.staff`. As a result, you can use schemas to create logical databases within a DB2 database.

To create a schema, use the `create schema` command.

**Who can use a schema?**

When you can create a schema, you can specify the owner of the schema using the `authorization` keyword; if you do not do so, the authorization ID that executed the `create schema` statement will be the owner of the schema. Privileges on the schema can also be granted to users or groups at the same time. (See [Part 1](#) in this series for more information on privileges.)

Once a schema exists, the owner of the schema can grant `CREATE_IN` privilege on the schema to other users or groups.

**Specifying the schema when creating an object**

The schema name for an object can be explicitly specified as follows:

```
create table DWAINE.table1 (c1 int, c2 int)
```

If the user `DWAINE` connects to the database `SAMPLE`, and issues the following statement:

```
create table t2 (c1 int)
```

The schema `DWAINE` is created (as long as `IMPLICIT_SCHEMA` has not been revoked from the user `DWAINE`), as well as the table in the database.

The ID used to connect to the database is known as the *authorization ID*.

**Specifying the schema when using DML commands**

When using DML commands (for example `select`, `insert`, `update`, `delete`) on database objects:



- The object schema can be explicitly specified on the object name, such as schema1.table1.
- The object schema can be specified using the `set current schema` or `set current sqlid` commands.
- If no object schema is explicitly specified, the schema will be set to the current authorization ID.

For example, if the user *DWAINE* connects to the database *SAMPLE* and issues the following statement:

```
select * from t2
```

DWAINE.T2 is selected from the table, if the table exists. Otherwise an error is returned.

## Table space states

### Determining a table space's state

To find the state for the table spaces in a database:

```
list tablespaces show detail
```

## Table space states

A table space can have a number of different states, as shown below.

0x0	Normal
0x1	Quiesced: SHARE
0x2	Quiesced: UPDATE
0x4	Quiesced: EXCLUSIVE
0x8	Load pending
0x10	Delete pending
0x20	Backup pending
0x40	Roll forward in progress
0x80	Roll forward pending
0x100	Restore pending
0x100	Recovery pending (not used)
0x200	Disable pending
0x400	Reorg in progress
0x800	Backup in progress
0x1000	Storage must be defined
0x2000	Restore in progress
0x4000	Offline and not accessible
0x8000	Drop pending
0x2000000	Storage may be defined
0x4000000	StorDef is in 'final' state
0x8000000	StorDef was changed prior to rollforward
0x10000000	DMS rebalancer is active
0x20000000	TBS deletion in progress
0x40000000	TBS creation in progress
0x8	For service use only

## Creating and manipulating various DB2 objects

### Introduction

This section discusses the purpose and use of:

- Buffer pools
- Table spaces
- Tables and indexes

- Views
- Identity columns
- Temporary tables
- Constraints
- Triggers

## Buffer pools

The database buffer pool area is a piece of memory used to cache a table's index and data pages as they are being read from disk to be scanned or modified. The buffer pool area helps to improve database system performance by allowing data to be accessed from memory instead of from disk. Because memory access is much faster than disk access, the less often that DB2 needs to read from or write to a disk, the better the system will perform.

When a database is created, one default buffer pool is created for the database. This buffer pool, IBMDEFAULTBP, has a page size of 4 KB and is sized depending on the operating system. For Windows, the default buffer pool is 250 pages or 1 MB; for UNIX, the default buffer pool is 1,000 pages or 4 MB. The default buffer pool cannot be dropped, but its size can be changed using the `alter bufferpool` command.

## Creating a buffer pool

The `create bufferpool` command has options to specify the following:

### Buffer pool name

Specifies the name of the buffer pool. The name cannot be used for any other buffer pools and cannot begin with the characters SYS or IBM.

### `immediate`

Specifies that the buffer pool will be created immediately if there is enough memory available on the system. If there is not enough reserved space in the database shared memory to allocate the new buffer pool, a warning is returned, and buffer pool creation will be DEFERRED, as described below. (`immediate` is the default.)

### `deferred`

Specifies that the buffer pool will be created the next time that the database is stopped and restarted.

### `all dbpartitionnums`

Specifies that the buffer pool will be created on all partitions in the database. This is the default if no database partition group is specified.

### `database partition group`

Specifies the database partition groups on which the buffer pool will be created. The buffer pool will be created on all database partitions that are part of the specified database partition groups.

### `size`

Specifies the size of the buffer pool and is defined in number of pages. In a partitioned database, this will be the default size for all database partitions where the buffer pool exists.

**numblockpages**

Specifies the number of pages to be created in the block-based area of the buffer pool. The actual value of `numblockpages` may differ from what was specified because the size must be a multiple of the `blocksize`. The block-based area of the buffer pool cannot be more than 98 percent of the total buffer pool size. Specifying a value of 0 will disable block I/O for the buffer pool.

**blocksize**

Specifies the number of pages within a given block in the block-based area of the buffer pool. The block size must be between 2 and 256 pages; the default value is 32 pages.

**pagesize**

Specifies the page size used for the buffer pool. The default page size is 4 KB or 4,096 bytes. The page size can be specified in either bytes or kilobytes.

**extended storage/not extended storage**

Specifies whether or not buffer pool victim pages will be copied to a secondary cache called *extended storage*. Retrieving data from extended storage is more efficient than retrieving it from disk but less efficient than retrieving it from the buffer pool, so it is not applicable to 64-bit environments.

Once a page size and name for a buffer pool have been defined, they cannot be altered.

**Example create bufferpool statements**

The following statement:

```
create bufferpool BP1 size 25000
```

creates a buffer pool named BP1 with a size of 100 MB (25,000 4 KB pages). Because the page size is not specified, the buffer pool uses the default page size of 4 KB. Since the `IMMEDIATE` option is the default, the buffer pool is allocated immediately and available for use as long as there is enough memory available to fulfill the request.

The following statement:

```
create bufferpool BP2 size 25000 pagesize 8 K
```

creates a buffer pool named BP2 with a size of 200 MB (25,000 8 KB pages). The buffer pool uses an 8 KB page size. Since the `immediate` option is the default, the buffer pool is allocated immediately and be available for use as long as there is enough memory available to fulfill the request.

The following statement:

```
create bufferpool BP3 deferred size 1000000
```

creates a buffer pool named BP3 with a size of 4 GB (1,000,000 4 KB pages). Because the page size is not specified, the buffer pool uses the default page size of 4 KB. Since the `deferred` option is specified, the buffer pool is not allocated until the database is stopped and restarted.

## Creating tables

To create a table in a database, you must first be connected to the database. You must also have SYSADM authority in the instance, or DBADM authority or createtab privilege in the database.

When creating a table, you can specify the following:

- Schema
- Table name
- Column definitions
- Primary/foreign keys
- Table space for the data, index, and long objects

The figure below shows an example.

```
Create table artists
(  artno          SMALLINT NOT NULL
,   name          VARCHAR (50) WITH DEFAULT 'abc'
,   classification CHAR (1)  NOT NULL
,   bio           CLOB (100K) LOGGED
,   article       DATALINK LINKTYPE URL FILE
,               LINK CONTROL MODE DB2OPTIONS,
,   picture       BLOB (2M)  NOT LOGGED COMPAT )
INDEX IN indtbsp
LONG IN longtbsp
IN datatbsp;
```

## Where are tables created?

If a table is created without the `in` clause, the table data (and its indexes and LOB data) is placed in the following order:

- In the IBMDEFAULTGROUP table space, if it exists and if the page size is sufficient
- In a user-created table space, which is of the smallest page size that is sufficient for the table
- In USERSPACE1, if it exists and has a sufficient page size

The `IN`, `INDEX IN`, and `LONG IN` clauses specify the table spaces in which the regular table data, index, and long objects are to be stored. Note that this only applies to DMS table spaces.

## Obtaining table information

You can obtain table information with the following commands:

Command	Description
<code>list tables</code>	List tables for the current user
<code>list tables for all</code>	List all tables defined in the database
<code>list tables for schema <i>schemaname</i></code>	List tables for the specified schema
<code>describe table <i>tablename</i></code>	Show the structure of the specified table

For example, the following command:

```
describe table department
```

produces the following output:

Column name	Type schema	Type name	Length	Scale	Nulls
DEPTNO	SYSIBM	CHARACTER	3	0	No
DEPTNAME	SYSIBM	VARCHAR	29	0	No
MGRNO	SYSIBM	CHARACTER	6	0	Yes
ADMRDEPT	SYSIBM	CHARACTER	3	0	No
LOCATION	SYSIBM	CHARACTER	16	0	Yes

## Indexes

An index can:

- Be ascending or descending (the default, if not specified, is ascending)
- Be unique or non-unique (the default, if not specified, is non-unique)
- Be compound
- Be used to enforce clustering
- Be bi-directional -- this is controlled by `allow` or `disallow reverse scans`
- Include additional columns -- This is only applicable for unique indexes.

Here are a number of `create unique` statements that illustrate these options:

```
create unique index itemno on albums (itemno) desc
create index clx1 on stock (shipdate) cluster allow reverse scans
create unique index incidx on stock (itemno) include (itemname)
create index item on stock (itemno) disallow reverse scans collect detailed statistics
```

## Identity columns

An *identity column* is a numeric column in a table that causes DB2 to automatically generate a unique numeric value for each row that is inserted into the table. A table can have a maximum of one identity column. The values for the column can be generated by DB2 *always* or *by default*:

- If values are always generated, the DB2 database always generates them, and applications are not allowed to provide an explicit value.
- If values are generated by default, then the values can be explicitly provided by an application; DB2 generates a value only if the application does not provide it. Thus, DB2 cannot guarantee the uniqueness of the values. This option is intended for data propagation, or loading and unloading of a table.

Let's look at an example. Given the table created using the following command:

```
create table inventory (partno INTEGER GENERATED ALWAYS AS IDENTITY
                      (START WITH 100 INCREMENT BY 1), description CHAR(20) )
```

And the following statements:

Statement	Result
insert into inventory VALUES (DEFAULT,'door') insert into inventory (description) VALUES ('hinge') insert into inventory VALUES (200,'windor') insert into inventory (description) VALUES ('frame')	inserts 100,door inserts 101,hinge error inserts 102,frame

Then the statement `SELECT * FROM inventory` gives the following:

```
100 door
101 hinge
102 frame
```

## Views

Views are derived from one or more base tables, nicknames, or views, and can be used interchangeably with base tables when retrieving data. When changes are made to the data shown in a view, the data is changed in the table itself. A view can be created to limit access to sensitive data, while allowing more general access to other data.

The data for a view is not stored separately from the table. In other words, a view uses no space in the database, other than its definition in the system catalogs.

The creator of a view needs to have at least `SELECT` privilege on the base tables referenced in the view definition.

The information about all existing views is stored in:

- SYSCAT.VIEWS
- SYSCAT.VIEWDEP
- SYSCAT.TABLES

The following `create view` statements show how views work:

```
create view DEPTSALARY AS SELECT DEPTNO, DEPTNAME, SUM(SALARY)
    AS TOTALS FROM PAYROLL GROUP BY DEPTNO,DEPTNAME
create view EMPSALARY AS SELECT EMPNO, EMPNAME, SALARY FROM PAYROLL,
    PERSONNEL WHERE EMPNO=EMPNUMB
```

## The with check option

The `with check option` specifies the constraint that every row that is inserted or updated through a view must conform to the definition of the view. A row that does not conform to the definition of the view is a row that does not satisfy the search conditions of the view.

For example, consider this command:

```
create view emp_view2 (empno, empname, deptno) AS (SELECT id, name,
    dept FROM employee WHERE dept = 10)with check option
```

When this view is used to insert or update with new values, the `with check option` restricts the input values for the *dept* column.

## Constraints

There are a number of types of constraints in DB2:

- Referential integrity constraints
- Unique constraints
- Check constraints
- Informational constraints

You cannot directly modify a constraint; you must instead drop it and create a new constraint with the characteristics you want.

Each constraint is discussed below.

### Referential integrity constraints

Referential integrity constraints are defined when a table is created, or subsequently using the `alter table` statement.

The clauses that establish referential integrity are:

- The primary key clause
- The unique constraint clause
- The foreign key clause
- The references clause

For example:

```
create table artists (artno INT, ... primary key (artno) foreign key dept (workdept)
references department on delete no action)
```

Let's take a look at the various referential integrity rules.

#### Insert rules:

- There is an implicit rule to back out of an insert if a parent is not found.

#### Delete rules:

- Restrict: A parent row is not deleted if dependent rows are found.
- Cascade: Deleting a row in a parent table automatically deletes any related rows in a dependent table.
- No Action (the default): Enforces presence of parent row for every child after all other referential constraints are applied.
- Set Null: Foreign key fields set to null; other columns left unchanged.

#### Update rules:

- Restrict: An update for a parent key will be rejected if a row in a dependent table matches the original values of key.

- No Action (the default): An update will be rejected for parent key if there is no matching row in the dependent table.

## Unique constraints

A unique constraint can be used as the primary key for a foreign key constraint, just like an explicitly declared primary key. This allows RI constraints (see [Referential integrity constraints](#)) to be placed on different columns within the same table.

A unique constraint forces the values in the column to be unique; the column cannot contain null values.

## Check constraints

A check constraint is used to enforce data integrity at the table level. It forces values in the table to conform to the constraint. All subsequent inserts and updates must conform to the defined constraints on the table, or the statement will fail.

If existing rows in the table do not meet the constraint, the constraint cannot be defined. Constraint checking can be turned off to speed up the addition of a large amount of data, but the table will be placed in CHECK PENDING state.

## Informational constraints

Informational constraints are rules that can be used by the optimizer, but are not enforced during runtime. Because other constraints may result in the overhead for insert, update, or delete operations, informational constraints may be a better alternative if the application already verifies the data.

Informational constraints can be:

- ENFORCED: The constraint is enforced by the database manager during normal operations such as insert, update, or delete
- NOT ENFORCED: When used, DB2 may return wrong results when any data in the table violates the constraint
- ENABLE QUERY OPTIMIZATION: The constraint can be used for query optimization under appropriate circumstances
- DISABLE QUERY OPTIMIZATION: The constraint can not be used for query optimization

## Triggers

A *trigger* defines a set of actions that are activated, or triggered, by an action on a specified base table. The actions triggered may cause other changes to the database or raise an exception. A trigger can be fired *before* or *after* inserts, updates, or deletes.

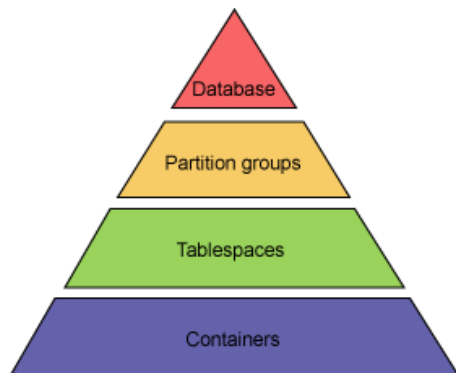
Triggers are used for:

- Validation, similar to constraints but more flexible.
- Conditioning, allowing new data to be modified or conditioned to a predefined value.
- Integrity, similar to referential integrity but more flexible.



# The DB2 storage triangle

## The storage triangle



Within a DB2 database, the storage of the data and indexes is defined and controlled at four different levels. To accommodate partitioned databases, there is an abstract layer called *partition groups*, as shown in the figure above. A partition group is a grouping or collection of one or more database partitions within a database. When a table space is created, it is assigned to a partition group and will only be created on the database partitions that are part of that partition group. Each table space must have one or more containers that define the physical storage for the table space. A container can be an operating system directory, a file with a predetermined size, a raw device such as an unformatted hard drive, a partition on the hard drive, or a logical volume.

## Table spaces

A *table space* is a logical entity used to define where tables and indexes will be stored within a database. As all DB2 tables and indexes reside in table spaces, this allows complete control over where the table and index data is physically stored.

A table space can be created using one or more underlying physical storage devices called *containers*. They provide the ability to create a physical database design that will provide optimal performance in any physical environment.

To get details about the table spaces in a database, use the following commands:

- `get snapshot for tablespaces`
- `list tablespaces`

## SMS table spaces

### Introduction

System Managed Space (SMS) table spaces use the file system manager provided by the operating system to allocate and manage the space where the tables are stored. Within an SMS table space, each container is an operating system directory, and table objects are created as files within that directory. When creating an SMS table space, the user must specify the name of the directory for each of the containers. DB2 will create the tables within the directories used in the table space by using unique file names for each object.

If a table space is created with more than one container, DB2 will balance the amount of data written to the containers. Since containers cannot be dynamically added to an SMS table space once it has been created, it is important to know the size requirements of the table space and create all required containers when the table space is created.

## Characteristics of SMS table spaces

With SMS table spaces:

- All table data and indexes share the same table space.
- Each table in a table space is given its own file name used by all containers. The file extension denotes the type of the data stored in the file.
- There is the possibility for dynamic file growth, with an upper boundary on size governed by the number of containers, operating system limits on the size of the file system, and operating system limits on size of individual files.
- When all space in a single container is allocated, the table space is considered full even if space remains in other containers.
- New containers can only be added to SMS on a partition that does not yet have any containers.
- On Linux or UNIX, the file system size may be increased.

SMS table spaces are very easy to administer, and are recommended for the TEMP table space.

## Creating SMS table spaces

To create an SMS table space use the following command:

```
create table space TS1 managed by system using ('path1', 'path2', 'path3')
```

When the path is specified for an SMS container, it can be either an absolute path or a relative path to the directory. If the directory does not exist, DB2 will create it. If the directory does exist, it cannot contain any files or subdirectories. For example, this command:

```
create table space ts1 managed by system using ('D:\DIR1')
```

specifies the absolute path to the directory. DB2 would create the `DIR1` directory on the `D:` drive on the database server if it does not already exist.

This command:

```
create tablespace ts2 managed by system using ('DIR1')
```

specifies the relative path `DIR1`. DB2 would create the `DIR1` directory under the database home directory.

The following SQL statements create an SMS table space with three containers on three separate drives or file systems. Note that the table space name is the same, as the examples are showing the differences between the UNIX/Linux and Windows table space definitions.

```
create tablespace smstbspc managed by system
    using ('d:\tbasp1', 'e:\tbasp2', 'f:\ tbasp3')
create tablespace smstbspc managed by system
    using ('/dbase/container1', '/dbase/container2', '/dbase/container3')
```

## Altering SMS table spaces

SMS table spaces can only be altered to change the prefetch size. Containers *cannot* be added to an SMS table space using the `alter` command. However, containers can be redefined, added, or removed during a redirected restore.

## Multi-dimensional clustering (MDC)

### Introduction to MDC

Multi-dimensional clustering (MDC) enables a table to be physically clustered on more than one key, or dimension, simultaneously. Prior to Version 8, DB2 supported only single-dimensional clustering of data using clustering indexes. When a clustering index is defined on a table, DB2 attempts to maintain the physical order of the data on pages, based on the key order of the clustering index, as records are inserted into and updated in the table. This can significantly improve the performance of queries that have predicates containing the keys of the clustering index because, with good clustering, only a portion of the physical table needs to be accessed. When the pages are stored sequentially on disk, more efficient prefetching can be performed.

With MDC, these same benefits are extended to more than one dimension, or clustering key. In the case of query performance, range queries involving any one or combination of the specified dimensions of the table will benefit from the underlying clustering. These queries will need to access only those pages having records with the specified dimension values, and the qualifying pages will be grouped together in extents.

A table with a clustering index can become unclustered over time, as available space is filled in the table. However, an MDC table is able to maintain its clustering over the specified dimensions automatically and continuously, eliminating the need to reorganize the table to restore the physical order of the data.

When an MDC table is created, the dimensional keys along which to cluster the table's data are specified. Each of the specified dimensions can be defined with one or more columns, the same as an index key. A dimension block index will be automatically created for each of the dimensions specified, and will be used to access data quickly and efficiently along each of the specified dimensions. A block index will also be automatically created, containing all dimension key columns. The block index will be used to maintain the clustering of the data during insert and update activity, as well as for quick and efficient access to the data.

Every unique combination of the table's dimension values forms a logical cell, which is physically comprised of blocks of pages, where a block is a set of consecutive pages on disk. The set of blocks that contain pages with data having the same key value of one of the dimension block indexes is called a *slice*. Every page of the table will be stored in only one block, and all blocks of the table will consist of the same number of pages, known as the *blocking factor*. The blocking

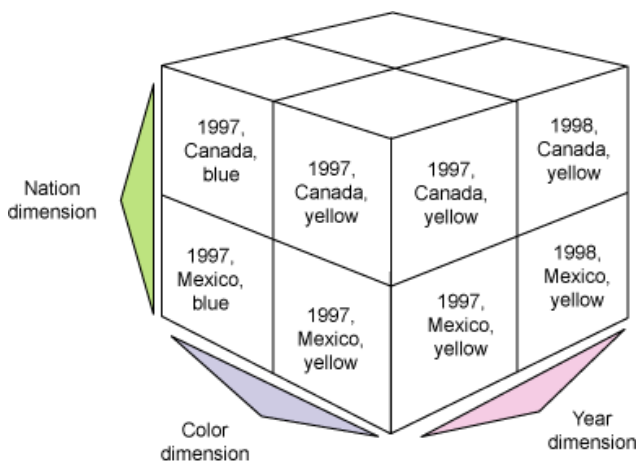
factor is equal to the table space's extent size, so that the block boundaries line up with extent boundaries.

## Creating an MDC table

To create an MDC table, you need to specify the dimensions of the table using the `organize by` parameter, as follows:

```
CREATE TABLE MDCTABLE(  
  Year INT,  
  Nation CHAR(25),  
  Colour VARCHAR(10),  
  ... )  
ORGANIZE BY(Year, Nation, Color)
```

In this example, the table will be organized on the year, nation, and color dimensions, and will logically look like the figure below.



You cannot alter a table and make it into an MDC table, so use the design advisor, if possible, before you create the database to see if your tables should be MDC tables or normal tables.

## MDC considerations

The following list summarizes the design considerations for MDC tables.

- When identifying candidate dimensions, search for attributes that are not too granular, thereby enabling more rows to be stored in each cell. This approach will make better use of block-level indexes.
- Higher data volumes may improve population density.
- It might be useful to load the data first as non-MDC tables for analysis only.
- The table space extent size is a critical parameter for efficient space usage.
- Although an MDC table may require a greater initial understanding of the data, the payback is that query times will likely improve.
- Some data may be unsuitable for MDC tables and would be better implemented using a standard clustering index.

- Although a smaller extent size will provide the most efficient use of disk space, the I/O for the queries should also be considered.
- A larger extent will normally reduce I/O cost, because more data will be read at a time. This, in turn, makes for smaller dimension block indexes because each dimension value will need fewer blocks. And, inserts will be quicker because new blocks will be needed less often.

## Table (range) partitioning

### Introduction to table partitioning

Several technologies in DB2 v8.2 (and prior releases) allow you to break up your data into smaller "chunks" for greater parallelism for queries, allow for partition elimination in queries, and in general help improve performance. As discussed in the previous section, MDC allows DB2 to place data on disk so that rows with the same value of the dimensional columns are stored together in blocks (a set of pages). Using this technique, a query that comes in looking for rows that have a particular dimension value will result in all other partitions being eliminated from the scan and only qualifying rows are accessed.

Similarly, the database partitioning feature can break up a set of tables such that a portion of the data resides on a database partition. The database partitions can reside on separate servers so a large scan can use the processing power of multiple servers.

DB2 V9 brings a new form of partition that is common in DB2 for z/OS and other data server vendor's products. Known as *table partitioning*, this feature allows you to take a single table and spread it across multiple tablespaces.

This new partitioning capability has a number of benefits, including a simplified syntax for creating them. Below is the simplified syntax for creating a partitioned table to store 24 months worth of data in 24 partitions spread across 4 tablespaces:

```
CREATE TABLE fact
(txn_id char(7), purchase_date date, ...)
IN tbsp1, tbsp2, tbsp3, tbsp4
PARTITION BY RANGE (purchase_date)
(
  STARTING FROM ('2005-01-01')
  ENDING ('2006-12-31')
  EVERY 1 MONTH
)
```

### Table partitioning syntax

The table creation syntax has a short and a long form. Before diving into the syntax, let's first talk about the partitioning column. Table partitioning allows for the specification of ranges of data, where each range goes into a separate partition. The partitioning column, or columns, defines the ranges. A partitioning column can be any of the DB2 base data types except for LOBs and LONG VARCHAR columns. You can also specify multiple columns (watch for an example later), and you can specify generated columns if you want to simulate partitioning on a function.

Here is the short form:

```
CREATE TABLE t1(c1 INT)
  IN tbsp1, tbsp2, tbsp3
  PARTITION BY RANGE(c1)
  (STARTING FROM (0) ENDING (80) EVERY (10))
```

Here is the same result using the long form:

```
CREATE TABLE t1(c1 INT)
  PARTITION BY RANGE(c1)
  (STARTING FROM (0) ENDING (10) IN tbsp1,
    ENDING (20) IN tbsp2,
    ENDING (30) IN tbsp3,
    ENDING (40) IN tbsp1,
    ENDING (50) IN tbsp2,
    ENDING (60) IN tbsp3,
    ENDING (70) IN tbsp1,
    ENDING (80) IN tbsp2)
```

## Quickly adding or removing data ranges

Another benefit is that when you *detach* a partition you get a separate table that contains the contents of that partition. You can detach a partition from a table and then do something with that newly detached partition, which is now a physical table. For example, you could archive off that table, move it to tertiary storage, copy it to another location, or whatever you want to do. DB2 V9 will asynchronously clean up any index keys on that partitioned table without impacting running applications.

Similar to adding a new partition, you simply create a table with the same definition as the partitioned table, load it with data, and then *attach* that partition to the main partitioned table, as follows:

```
ALTER TABLE FACT_TABLE ATTACH PARTITION
  STARTING '06-01-2006'
  ENDING '06-30-2006'
  FROM TABLE FACT_NEW_MONTH
```

## Table compression

### Introduction to table compression

Row compression works by looking at the contents of the entire table, finding repeating byte strings, storing those strings in a dictionary, and then replacing those strings that appear in the table with a symbol that represents the actual data stored in the dictionary. The major benefit is that DB2 is looking at all the data in the table and at the data row in its entirety -- not just repeating column values. For example, if there is a repeating substring within a column, that can be compressed out. If there is a repeating string that spans columns (like city,state), that can also be compressed out into a single symbol.

### Using table compression

To use row compression you must first set the table to be compression eligible, and then you must generate the dictionary that contains the common strings from within the table. To set the table to be eligible for compression, use either of the following commands:

```
create table table_name ... compress yes  
or  
alter table tablename compress yes
```

## Creating the compression dictionary

Creating the compression dictionary allows the table to be compressed. DB2 then needs to scan the data in the table to find the common strings that it can compress out of the table and put in the dictionary. To do this you use the `reorg` command. The first time you compress a table (or if you want to rebuild the compression dictionary) you must run the command:

```
reorg table table_name resetdictionary
```

This will scan the table, create the dictionary, and then perform the actual table reorganization, compressing the data as it goes. From this point onward, any insert into this table or subsequent load of data will honor the compression dictionary and compress all new data. If in the future you want to run a normal table reorg and not rebuild the dictionary, you can run the command:

```
reorg table table_name keepdictionary
```

Each table has its own dictionary, meaning that a partitioned table will have a separate dictionary for each partition. This is good because it allows DB2 to adapt to changes in the data as you roll in a new partition.

## Estimating space savings

If you want to just see how much space you can save without actually compressing the table, you can do that too. The `DB2 INSPECT` command now has an option to report the number of pages saved if you decide to compress a given table. The syntax for this is:

```
db2 inspect rowcompestimate table name table_name results keep file_name
```

You then run the command:

```
db2inspf file_nameoutput_file_name
```

to convert the binary output file of inspect into a readable text file called *output\_file\_name*. This file contains the estimated percentage of data pages saved from compression.

## Table compression steps for a new table

If you are starting with a new system, you may want to:

- Create the table with compression `yes`.
- Load a representative sample of the data into the table.
- Reorg the table with `resetdictionary` to create a new dictionary.
- Load the rest of the data into the table (this load will respect the dictionary and compress on the fly as it loads).

# XML

## Introduction to XML in DB2

You've been able to store XML data in DB2 for quite some time. Of course, you can store the object as a CLOB and, with the XML extender, you can shred the document into relational tables that let you efficiently access subcomponents of an XML document with a query. But each of these methods has a disadvantage. Shredded documents can result in loss of document fidelity and make it difficult to change the XML schema. The biggest benefit of XML is that the schema is completely flexible, so making it rigid by shredding it into relations is counterproductive. With CLOB you can keep the flexibility but every time you want to read components of the XML you need to parse the CLOB at runtime, so performance is poor.

DB2 V9 introduces a completely new XML storage engine where XML data is stored hierarchically. XML is hierarchical in nature, so storing the XML hierarchically in the engine preserves fidelity, allows for flexible schema, and also delivers high performance sub document access. This new hierarchical storage engine sits inside the same DB2 data server as the relational engine, so now you can store customer information alongside their XML purchase orders and search all of the information efficiently.

## XML columns in DB2 tables

XML is stored inside DB2 in a hierarchical format. XML itself is hierarchical starting from the root tag (or node) and then traversing through the XML string (or document). In DB2 the XML is stored inside of data pages in this hierarchical structure. If the XML data is larger than a single data page, then the XML tree is broken up into subtrees with each subtree stored on a data page and the pages linked together.

To create a table with XML data simply run the command:

```
create table table_name (col1 data_type, ..., xml_col_name XML)
```

This allows you to create the table with your relational columns as you would today, and for your XML information you just assign the column a datatype of XML. Now you can store the XML data in that column.

## XML indexes

Creating an index is similar to creating a typical index on relational data with the exception that you are not indexing a *column*, but rather a component of the XML schema defined in the above *xml\_column\_name* column. The syntax would look something like the following:

```
create index index_name on table_name (xml_column_name)  
generate key using xmlpattern '/po/purchaser/@pname' as sql varchar(50)
```

For more information about XML support in DB2 9, see the [Resources](#) section.



## Conclusion

### Summary

In this tutorial we discussed:

- Creating databases
- Using schemas
- The different table space states
- Creating and manipulating DB2 objects
- Creating SMS table spaces and their characteristics
- The characteristics and use of DB2's automatic storage
- Implementing table partitioning and MDC
- Using table compression
- Using XML

You are now ready to take the data placement portion of the DBA certification test. Good Luck!

© Copyright IBM Corporation 2006

([www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml))

Trademarks

([www.ibm.com/developerworks/ibm/trademarks/](http://www.ibm.com/developerworks/ibm/trademarks/))