This tutorial discusses the creation of IBM DB2® databases, as well as various methods used for placing and storing objects within a database. The focus is on partitioning, compression, and XML, which are all important performance and application development concepts you need to store and access data quickly and efficiently. This is second in a series of eight tutorials you can use to help you prepare for the DB2 10.1 DBA for Linux®, UNIX®, and Windows® certification exam 611. The material here primarily covers the objectives in Section 2 of the exam.

View more content in this series

Before you start

About this series

If you are preparing to take the DB2 10.1 for Linux, UNIX, and Windows DBA certification exam (exam 611), you’ve come to the right place — a study hall of sorts. This series of eight DB2 certification preparation tutorials covers the major concepts you'll need to know for the test.

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. The focus is on partitioning, compression, and XML, which are all important performance and application development concepts you need to know to store and access data quickly and efficiently. This is the second in a series of eight tutorials that you can use to help you prepare for the DB2 10.1 for Linux, UNIX, and Windows DBA certification exam (exam 611). The material here primarily covers the objectives in Section 2 of the exam (“Physical design”).

Objectives

After completing this tutorial, you should be able to:

• Demonstrate the ability to create a database and manipulate various DB2 objects.
- Know how to convert existing database to automatic storage database.
- Use the Admin_move_table feature.
- Demonstrate knowledge of partitioning capabilities.
- Demonstrate knowledge of XML data objects.
- Demonstrate knowledge of compression.
- Demonstrate knowledge of new table features.
- Knowledge of Multi-temperature data feature.

**Prerequisites**

To take the DB2 10.1 DBA exam 611, you must have already passed the DB2 10.1 Fundamentals exam 610. We recommend that you study the DB2 10.1 fundamentals certification exam 610 series before starting this series.

To help you understand some of the material presented here, you should be familiar with the following terms:

- **Object**— Anything in a database that can be created or manipulated with SQL (e.g., tables, views, indices, 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 about 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.

**System requirements**

You do not need a copy of DB2 to complete this tutorial. However, you will get more out of it if you download the free trial version of IBM DB2 10.1 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 cataloged for this instance. Databases are implicitly cataloged when the create database command is issued, and can also be explicitly cataloged with catalog database.

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 is done for you:

- Setting up of all the system catalog tables 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:

- Whether the database should be automatically configured
- One or more storage paths
- Drive or path on which to create the database
- An alias for the database in the system database directory
- Code set and territory
- Collating sequence
- Default page size
- default extent size of table spaces in the database
- Table space definitions for the CATALOG, TEMPORARY, and USERSPACE1 table spaces

The default database

Each new database has:

- A default buffer pool defined (IBMDEFAULTBP).
- A default storage group named IBMSTOGROUP, which is automatically created while creating AUTOMATIC STORAGE database and the database storage paths are automatically assigned to that storage group.
- 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 is 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, indices, and packages, for example) and security information about the type of access 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 DFTDDBPATH 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 (DB2, for example) 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 created on the drive or directory. For example, imagine we have three databases, DB2QS, TEST and SAMPLE, that were all created on the C drive on Windows. There will be three directories: SQL00001, SQL00002, and SQL00003.

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.
Figure 1. Directory under which database is created

In the example above, the database DB2QS was created in the SQL00001 directory, the database TEST was created in the SQL00002 and the database SAMPLE was created in the SQL00003 directory under the NODExxxx directory.

For AUTOMATIC STORAGE databases:

- The system catalog table space (SYSCATSPACE) will use the directory T0000000.
- The system temporary table space (TEMPSPACE1) will use the directory T0000001.
- The default user table space (USERSPACE1) will use the directory T0000002.

EXAMPLE: The following command on windows: `db2 create db AUTODB2 on D:\DB2\Data dbpath on D:\DB2` will create a folder structure as specified in the next figure.
Create database commands for Linux/UNIX

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/sqldbdir`
- `/database/dbinst/NODE0000/SQL00001`

Create database commands for Windows

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\sqldbdir`
- `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 command 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, 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 Linux/UNIX command create database SAMPLE on /mydbs collate using identity or Windows command 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.

Creating and manipulating various DB2 objects

This section discusses the purpose and use of the following DB2 objects:

- Data placing objects
  - Buffer pools
  - Table spaces
- Application objects
  - Schema
  - Tables
  - Indices
  - Identity columns
  - Views
  - 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 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, will have a page size equal to the database page size as specified at the database creation time. If no default page size was specified at database creation, the default buffer pool will have a page size of 4 KB. The default buffer pool size depends on the operating system. For Windows, the default buffer pool is 250 pages or 1 MB; for UNIX, it's 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.

• **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 block size. 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 bytes or kilobytes.

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

**create bufferpool statement**

The following statement 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: `create bufferpool BP1 size 25000`.

The following statement 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 available for use as long as there is enough memory available to fulfill the request: `create bufferpool BP2 size 25000 pagesize 8 K`.

The following statement creates a buffer pool named BP3 with a size of 4 GB (1 million 4-KB pages). Because the page size is not specified, the buffer pool uses the default page size of 4 KB; 128 pages of the buffer pool will be used in the block-based area and blocks of the buffer pool will have 64 pages each. Since the `deferred` option is specified, the buffer pool is not allocated until the database is stopped and restarted: `create bufferpool BP3 deferred size 1000000 NUMBLOCKPAGES 128 BLOCKSIZE 64`.
Table spaces

A table space is a storage structure containing tables, indices, large objects, and long data. They are used to organize data in a database into logical storage groupings that relate to where data is stored on a system. Table spaces are stored in database partition groups.

Schema

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, indices, 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. This is done by defining a schema using the \texttt{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 is created with every database and placed into the SYSCATSPACE table space:

- \texttt{SYSIBM} — The base system catalogs; direct access is not recommended.
- \texttt{SYSCAT} — SELECT authority granted to PUBLIC on this schema; catalog read-only views; recommended way to obtain catalog information.
- \texttt{SYSSTAT} — Updatable catalog views; influences the optimizer.
- \texttt{SYSFUN} — User-defined functions.

How is a schema used in DB2?

Use a schema to fully qualify a table or other object name: \texttt{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 \texttt{create schema} command.

Who can use a schema?

When you can create a schema, you can specify the owner of the schema using \texttt{authorization}. If you do not, the authorization ID that executed the \texttt{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.

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 DWAINETABLE1 (c1 int, c2 int)`.

If the user DWAINETABLE connects to the database SAMPLE and issues the following statement: `create table t2 (c1 int)`, the schema DWAINETABLE is created (as long as IMPLICIT_SCHEMA has not been revoked from the user DWAINETABLE), 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 user DWAINETABLE connects to the database SAMPLE and issues the statement `select * from t2`, DWAINETABLE.T2 is selected from the table, if the table exists. Otherwise, an error is returned.

Tables

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 listing below shows an example:

**Listing 1. Example of table creation**

```
CREATE TABLE EMPLOYEE_SALARY ( DEPTNO CHAR(3) NOT NULL, DEPTNAME VARCHAR(36) NOT NULL, EMPNO CHAR(6) NOT NULL, SALARY DECIMAL(9,2) NOT NULL WITH DEFAULT) INDEX IN indtbsp IN datatbsp
```

Where are tables created?

If a table is created without the `in` clause, the table data (and its indices 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 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 commands in the table below.

<table>
<thead>
<tr>
<th>Table 1. Obtaining table information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>list tables</td>
</tr>
<tr>
<td>list tables for all</td>
</tr>
<tr>
<td>list tables for schema schemaname</td>
</tr>
<tr>
<td>describe table tablename</td>
</tr>
</tbody>
</table>

For example, the command `describe table department` produces the following output.

**Figure 3. Table information output**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
<th>Length</th>
<th>Scale</th>
<th>Nulls</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTNO</td>
<td>SBIM</td>
<td>3</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>DEPTNAME</td>
<td>SBIM</td>
<td>29</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>MGRNO</td>
<td>SBIM</td>
<td>6</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>ADMNDEPT</td>
<td>SBIM</td>
<td>3</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>LOCATION</td>
<td>SBIM</td>
<td>16</td>
<td>0</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Indices**

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 indices.

Following 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 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, 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))
```

**HINT:** If you plan to load or import data into your table, use the `GENERATED BY DEFAULT IDENTITY`, which will make use of the supplied values if they are provided; if the data is missing or explicitly NULL, a value is generated.

**Table 2. Identity column statement and values**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO inventory VALUES (DEFAULT, 'door')</td>
<td>inserts 100, door</td>
</tr>
<tr>
<td>INSERT INTO inventory (description) VALUES ('hinge')</td>
<td>inserts 101, hinge</td>
</tr>
<tr>
<td>INSERT INTO inventory VALUES (200, 'windor')</td>
<td>error</td>
</tr>
<tr>
<td>INSERT INTO inventory (description) VALUES ('frame')</td>
<td>inserts 102, frame</td>
</tr>
</tbody>
</table>

Then the statement `SELECT * FROM inventory` gives `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. All views can be used just like tables for data retrieval. Whether a view can be used in an insert, update, or delete operation depends on its definition. Views are classified by the operations they allow:

• Deletable
• Updatable
• Insertable
• Read-only

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
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=EMPLUMB`

**The with check option**

The `with check option` specifies the constraint that every row 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 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 cannot 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.

**Table spaces**

A *table space* is a storage structure containing tables, indices, large objects, and long data. They are used to organize data in a database into logical storage groupings that relate to where data is stored on a system. Table spaces are stored in database partition groups.

Table spaces consist of one or more containers. A container can be a directory name, a device name, or a file name. A single table space can have several containers. It is possible for multiple containers (from one or more table spaces) to be created on the same physical storage device (although you will get the best performance if each container you create uses a different storage device).

**Figure 4. Table spaces and tables in a database**

If you are using automatic storage table spaces, the creation and management of containers is handled automatically by the database manager. If you are not using automatic storage table spaces, you must define and manage containers yourself. To get details about the table spaces in a database, use the following commands: `get snapshot for tablespaces` or `list tablespaces`.

**SMS table spaces**

*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 indices 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, OS limits on the size of the file system, and OS 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 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, `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.

The command `create table space 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 Linux/UNIX and Windows table space definitions:

```sql
create tablespace smstbspc managed by system using ('d:\tbspc1', 'e:\tbspc2', 'f:\tbspc3')
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.

DMS table spaces

In a database managed space (DMS) table space, the database manager controls the storage space. Unlike SMS table spaces, storage space is pre-allocated on the file system based on container definitions you specify when you create the DMS table space.

The DMS storage model consists of a limited number of files or devices where space is managed by the database manager. You decide which files and devices to use when creating containers, and you manage the space for those files and devices.

A DMS table space containing user defined tables and data can be defined as a large (the default) or regular table space that stores any table data or index data.

Characteristics of DMS table spaces

With DMS table spaces:

- The database manager uses striping to ensure an even distribution of data across all containers.
- Containers that make up a DMS table space are not required to be the same size. If any container is full, DMS table spaces use available free space from other containers.
- Because space is pre-allocated, it must be available before the table space can be created.
- Enabling DMS table spaces that use file containers for automatic resizing allows the database manager to handle the full table space condition automatically by extending existing containers for you.
- The table space is considered to be full when all of the space within the containers has been used.
- Unlike for SMS table spaces, you can add or extend containers manually, using the `ALTER TABLESPACE` statement, allowing more storage space to be given to the table space.

Creating DMS table spaces

The following statement creates a DMS table space without enabling auto-resize (the default):

```
CREATE TABLESPACE DMS1 MANAGED BY DATABASE USING FILE '/db2files/DMS1' 10 M)
```

To enable the auto-resize feature, specify the `AUTORESIZE YES` clause for the `CREATE TABLESPACE` statement:

```
CREATE TABLESPACE DMS1 MANAGED BY DATABASE USING (FILE '/db2files/DMS1' 10 M) AUTORESIZE YES
```

To create a table space that can grow to 100 MB (per database partition if the database has multiple database partitions), use the `MAXSIZE` clause of the `CREATE TABLESPACE` statement:
CREATE TABLESPACE DMS1 MANAGED BY DATABASE USING (FILE '/db2files/DMS1' 10 M) AUTORESIZE YES MAXSIZE 100 M

If you do not specify the `MAXSIZE` clause, there is no maximum limit when the auto-resize feature is enabled. The table space will grow until a file system limit is reached.

The `INCREASESIZE` clause of the `CREATE TABLESPACE` statement defines the amount of space used to increase the table space when there are no free extents within the table space. You can specify the value as an explicit size or as a percentage, as shown in the following examples:

### Listing 2. Specify value as explicit size or percentage

```sql
CREATE TABLESPACE DMS1 MANAGED BY DATABASE USING (FILE '/db2files/DMS1' 10 M) AUTORESIZE YES INCREASESIZE 5 M
CREATE TABLESPACE DMS1 MANAGED BY DATABASE USING (FILE '/db2files/DMS1' 10 M) AUTORESIZE YES INCREASESIZE 50 PERCENT
```

### Altering DMS table spaces

You can enable or disable the `auto-resize` feature after creating a DMS table space by using `ALTER TABLESPACE` statement with the `AUTORESIZE` clause:

```sql
ALTER TABLESPACE DMS1 AUTORESIZE YES
ALTER TABLESPACE DMS1 AUTORESIZE NO
```

Use the `ALTER TABLESPACE` statement to change the value of `MAXSIZE` for a table space that has auto-resize already enabled, as shown in the following examples:

```sql
ALTER TABLESPACE DMS1 MAXSIZE 1 G
ALTER TABLESPACE DMS1 MAXSIZE NONE
```

You can also define the amount of space used to increase the table space when there are no free extents as shown in the following examples:

```sql
ALTER TABLESPACE DMS1 INCREASESIZE 5 M
ALTER TABLESPACE DMS1 INCREASESIZE 50 PERCENT
```

### Automatic storage

#### What is automatic storage?

Automatic storage allows you to specify one or more storage paths for a database. Then when you create table space, a container file will be created automatically on each storage path by DB2. You can enable or configure automatic storage for a database when it is created, as follows:

```sql
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: `db2 alter database db_name add storage on db_path3`. You can also drop a storage path from a database set up for automatic storage using the `drop storage` parameter: `db2 alter database db_name drop 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): \texttt{db2 create tablespace ts\_name}. Or you can create a table space
and specify its initial size and growth characteristics:

\begin{verbatim}
db2 create tablespace ts\_name initialsize 10M increasesize 10M maxsize 100M
\end{verbatim}

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

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:

\begin{verbatim}
db2 create tablespace ts\_name managed by automatic storage
\end{verbatim}

**Storage groups**

A storage group is a named set of storage paths where data can be stored. Storage groups are
configured to represent different classes of storage available to your database system. You can
assign table spaces to the storage group that best suits the data. Only automatic storage table
spaces use storage groups.

A table space can be associated with only one storage group, but a storage group can have
multiple table space associations. To manage storage group objects, you can use the \texttt{CREATE
STOGROUP}, \texttt{ALTER STOGROUP}, \texttt{RENAME STOGROUP}, and \texttt{DROP}
statements.

With the table partitioning feature, you can place table data in multiple table spaces. Using this
feature, storage groups can store a subset of table data on fast storage while the remainder of the
data is on one or more layers of slower storage. Use storage groups to support multi-temperature
storage, which prioritizes data based on classes of storage. For example, you can create storage
groups that map to the different tiers of storage in your database system. Then the defined table
spaces are associated with these storage groups. More information about multi-temperature
storage is available in the "Multi-temperature data feature" section.

**The default storage groups**

When you create a database, a default storage group named IBMSTOGROUP is automatically
created. However, a database created with the \texttt{AUTOMATIC STORAGE NO} clause does not have
a default storage group. The first storage group created with the \texttt{CREATE STOGROUP} statement
becomes the designated default storage group. There can only be one storage group designated
as the default storage group.

If a database has storage groups, the default storage group is used when an automatic storage
managed table space is created without explicitly specifying the storage group.

You can designate a default storage group by using either the \texttt{CREATE STOGROUP} or \texttt{ALTER STOGROUP}
statement. When you designate a different storage group as the default storage group, there is no
impact to the existing table spaces using the old default storage group. To alter the storage group
associated with a table space, use the ALTER TABLESPACE statement. You can determine which storage group is the default storage group by using the SYSCAT.STOGROUPS catalog view.

You cannot drop the current default storage group. You can drop the IBMSTOGROUP storage group if it is not designated as the default storage group at that time. If you drop the IBMSTOGROUP storage group, you can create another storage group with that name.

Creating storage groups

To create a storage group by using the command line, enter CREATE STOGROUP operational_sg ON '/filesystem1', '/filesystem2', '/filesystem3'... where operational_sg is the name of the storage group and /filesystem1, /filesystem2, e/filesystem3 , ... are the storage paths to be added.

Altering storage groups

You can use the ALTER STOGROUP statement to alter the definition of a storage group, including setting media attributes, setting a data tag, or setting a default storage group. You can also add and remove storage paths from a storage group.

To add storage paths '/hdd/path1' and '/hdd/path2' to storage group sg, issue the following ALTER STOGROUP statement: ALTER STOGROUP sg ADD '/hdd/path1', '/hdd/path2'. To drop storage paths '/db2/filesystem1' and '/db2/filesystem2' from storage group sg, issue the following ALTER STOGROUP statement: ALTER STOGROUP sg DROP '/db2/filesystem1', '/db2/filesystem2'.

How to convert existing database to automatic storage

You can convert an existing non-automatic storage database to use automatic storage by using the CREATE STOGROUP statement to define the default storage group within a database.

When you define a storage group for a database, existing table spaces are not automatically converted to use automatic storage. By default, only future table spaces you create are automatic storage table spaces. You must use the ALTER TABLESPACE statement to convert existing table spaces to use automatic storage.

Example 1: Converting a database on Linux or UNIX

Assume that a database is a non-automatic storage database and that /data1/as and /data2/as are the paths you want to use for automatic storage table spaces. To convert the database to an automatic storage database, create a storage group with /data1/as and /data2/as as paths: CREATE STOGROUP sg ON '/data1/as', '/data2/as'.

Example 2: Converting a database on Windows operating systems

Assume that a database is a nonautomatic storage database, and that F:\DB2DATA and G: are the paths you want to use for automatic storage table spaces. To convert the database to an automatic storage database, create a storage group with F:\DB2DATA and G: as paths: CREATE STOGROUP sg ON 'F:\DB2DATA', 'G:'. After creating the default storage group, to convert an existing DMS table space tbspc1 to use automatic storage, issue the following statements:
ALTER TABLESPACE tspc1 MANAGED BY AUTOMATIC STORAGE USING STOGROUP sg ALTER TABLESPACE tspc1 REBALANCE

The `rebalance` operation moves data from the non-automatic storage containers to the new automatic storage containers.

**How to check if an existing database is an automatic storage**

There are several ways to check if an existing database is an automatic storage database. You can use any of the following options.

- The `ADMIN_GET_STORAGE_PATHS` table function — It returns a list of automatic storage paths for each database storage group, including file system information for each storage path.
- The `SYSIBMADM.SNAPSTORAGE_PATHS` administrative view — It returns a list of automatic storage paths for the database including file system information for each storage path.
- The `db2look` utility — Generates the DDL statements for a database by object type.

**Using Admin_move_table feature**

You can move tables online and offline using the `ADMIN.Move_TABLE` procedure. In fact, this can be used to move the data in a table to a new table object of the same name (but with possibly different storage characteristics such as a different table space) while the data remains online and available for access. You can also generate a new optimal compression dictionary when a table is moved.

The `ADMIN.Move_TABLE` stored procedure creates a protocol table comprising rows with status information and configuration options related to the table to be moved. The return set from this procedure is the rows from that protocol table related to the table to be moved.

You can invoke the `ADMIN.Move_TABLE` in one of two ways:

**Method 1** — Modify only certain parts of the table definition for the target table. Fill out the `data_tbsp`, `index_tbsp`, and `lob_tbsp` parameters while calling the procedure, leaving the other optional parameters blank.

**Example**

This example calls the stored procedure using the first method, where the target table is defined within the procedure, to move a table named T1, located in the schema SCHEMA1. Additionally, the column definitions of the target table are passed to the procedure.

**Listing 3. Example for the stored procedure using the first method**

```sql
CALL SYSPROC.ADMIN.Move_TABLE( 'SCHEMA1', 'T1', 'ACCOUNTING', 'ACCOUNT_IDX', 'ACCOUNT_LONG', '', '', '',
'CUSTOMER VARCHAR(80), REGION CHAR(5), YEAR INTEGER, CONTENTS CLOB', '', 'MOVE')
```

The above example moves the table SCHEMA1.T1 to a new table with the same name that has the column definitions ('CUSTOMER VARCHAR(80), REGION CHAR(5), YEAR INTEGER,
CONTENTS CLOB) and resides in table space 'ACCOUNTING' with its indices table space 'ACCOUNT_IDX' and its LOBs table space 'ACCOUNT_LONG'.

**Method 2**— create the target table and provide its name to the procedure. This provides you with more control and flexibility by allowing you to create the target table beforehand, rather than having the stored procedure create it.

**Example**

This example is equivalent to the previous one, but it calls the stored procedure using the second method, where the target table is created outside the procedure and is named within the target_tabname parameter, to move the same table as in the previous example.

**Listing 4. Example for the stored procedure using the second method**

```sql
CREATE TABLE SCHEMA1.T1_TGT ( CUSTOMER VARCHAR(80), REGION CHAR(5), YEAR INTEGER, CONTENTS CLOB) IN ACCOUNTING INDEX IN ACCOUNT_IDX LONG IN ACCOUNT_LONG'

CALL SYSPROC.ADMIN_MOVE_TABLE( 'SCHEMA', 'T1', 'T1_TGT', '', 'MOVE')
```

For online data movement:

1. The procedure creates a shadow table to which the data are copied.
2. Any changes to the source table during the copy phase are captured using triggers and placed in a staging table.
3. After the copy phase is completed, the changes captured in the staging table are replayed to the shadow copy.
4. Following that, the stored procedure briefly takes the source table offline and assigns the source table name and index names to the shadow copy and its indices.
5. The shadow table is then brought online, replacing the source table. By default, the source table is dropped, but you can use the `KEEP` option to retain it under a different name.

Obviously, the online operation costs more server resources (disk space and processing power), so make sure you only use it if you value availability more than cost, space, move performance, and transaction overhead. Additionally, avoid performing online moves for tables without indices, particularly unique ones as it might result in deadlocks and complex or expensive replay.

**Handling online move failure**

If the online move fails, rerun it:

1. Fix the problem that caused the table move to fail.
2. Determine the stage that was in progress when the table move failed by querying the SYSTOOLS.ADMIN_MOVE_TABLE protocol table for the status.
3. Call the stored procedure again, specifying the applicable option:
   - If the status of the procedure is INIT, use the `INIT` option.
   - If the status of the procedure is COPY, use the `COPY` option.
   - If the status of the procedure is REPLAY, use the `REPLAY` or `SWAP` option.
If the status of the procedure is CLEANUP, use the CLEANUP option.

You can cancel the move by specifying the CANCEL option for the stored procedure if the status of an online table move is not COMPLETED or CLEANUP.

**Table space states**

**Determining a table space's state**

To find the state for the table spaces in a database: `list tablespaces show detail`. A table space can have a number of different states, as shown below.

**Table 3. Table space states**

<table>
<thead>
<tr>
<th>State</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0x0</td>
</tr>
<tr>
<td>Quiesced share</td>
<td>0x1</td>
</tr>
<tr>
<td>Quiesced update</td>
<td>0x2</td>
</tr>
<tr>
<td>Quiesced exclusive</td>
<td>0x4</td>
</tr>
<tr>
<td>Backup pending</td>
<td>0x20</td>
</tr>
<tr>
<td>Roll-forward in progress</td>
<td>0x40</td>
</tr>
<tr>
<td>Roll-forward pending</td>
<td>0x80</td>
</tr>
<tr>
<td>Restore pending</td>
<td>0x100</td>
</tr>
<tr>
<td>Disable pending</td>
<td>0x200</td>
</tr>
<tr>
<td>Reorg in Progress</td>
<td>0x400</td>
</tr>
<tr>
<td>Backup in Progress</td>
<td>0x800</td>
</tr>
<tr>
<td>Storage Must be Defined</td>
<td>0x1000</td>
</tr>
<tr>
<td>Restore in Progress</td>
<td>0x2000</td>
</tr>
<tr>
<td>Offline and Not Accessible</td>
<td>0x4000</td>
</tr>
<tr>
<td>Drop Pending</td>
<td>0x8000</td>
</tr>
<tr>
<td>Suspend Write</td>
<td>0x10000</td>
</tr>
<tr>
<td>Load in Progress</td>
<td>0x20000</td>
</tr>
<tr>
<td>Storage May be Defined</td>
<td>0x20000000</td>
</tr>
<tr>
<td>DMS Rebalance in Progress</td>
<td>0x10000000</td>
</tr>
<tr>
<td>Table Space Deletion in Progress</td>
<td>0x20000000</td>
</tr>
<tr>
<td>Table Space Creation in Progress</td>
<td>0x40000000</td>
</tr>
</tbody>
</table>

More details about each of the above states are available at [Information Center](https://www.ibm.com/support/knowledgecenter/SSSHTN_10.1.0/com.ibm.db2.luw.admin.dbapp.doc/doc/csp_tablespace.html).

**Data organization and partitioning**

**Table (range) partitioning**

Several technologies in DB2 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. One of these technologies is the Table (or Range) Partitioning feature, this feature allows you to take a single table and divide it into partitions which may be spread across multiple table spaces.

This partitioning capability has a number of benefits, including:

1. Simplified syntax for creating table or range partitions.
2. Large tables can be spread across several table spaces. Note that all of the table spaces specified must have the same page size, extent size, storage mechanism (DMS or SMS), and type (REGULAR or LARGE), and all of the table spaces must be in the same database partition group.
3. Simplified process for rolling in and rolling out of table data by seamlessly attaching and detaching table partitions.
4. Improved query performance by automatically eliminating data partitions based on predicates in the query.

Below is the simplified syntax for creating a partitioned table to store 24 month's worth of data in 24 partitions spread across four table spaces.

**Listing 5. Syntax for creating a partitioned table**

```
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 to simulate partitioning on a function. Here is the short form.

**Listing 6. Short form for table partitioning syntax**

```
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.

**Listing 7. Long form for table partitioning syntax**

```
CREATE TABLE t1(c1 INT) PARTITION BY RANGE(c1) (STARTING FROM (0) ENDING (80) EVERY (10)) IN tbsp1, tbsp2, tbsp3, tbsp4
```

**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 10.1
will asynchronously clean up any index keys on that partitioned table without impacting running
applications.

In fact, detaching a table partition from a partitioned table is invoked using the ALTER
TABLE..DETACH PARTITION...INTO clause and consists of two phases:

1. The partition is logically detached from the partitioned table.
2. Asynchronously the logically detached partitioned is converted into a stand-alone table.

This implies that detached data becomes invisible instantly and no actual data movement
occurs during the detach process. A typical example of detaching a table partition: ALTER TABLE
part_table DETACH PARTITION p0 INTO TABLE p0_tab;

In the above statement, the partition p0 is detached from the part_table table, and asynchronously
a new physical table p0_tab is created containing the content of the partition p0.

Note that the following conditions must be met before you can perform a DETACH PARTITION
operation:

- The table to be detached from (source table) must exist and be a partitioned table.
- The data partition to be detached must exist in the source table.
- The source table must have more than one data partition. A partitioned table must have at
  least one data partition. Only visible and attached data partitions pertain in this context. An
  attached data partition is a data partition that is attached but not yet validated by the SET
  INTEGRITY statement.
- The name of the table to be created by the DETACH PARTITION operation (target table) must
  not exist.
- DETACH PARTITION is not allowed on a table that is the parent of an enforced referential
  integrity (RI) relationship. So you must change the RI constraints into informational by setting
  them to NOT ENFORCED before you can detach a partition. Then you can enforce the constraint
  again after the detaching process ends.
- If there are any detached dependent tables (tables need to be incrementally maintained with
  respect to the detached data partition), the SET INTEGRITY statement is required to be run on
  these tables to incrementally maintain the tables. With DB2 9.7 Fix Pack 1 or later, after the
  SET INTEGRITY statement is run on all detached dependent tables, the asynchronous partition
detach task makes the data partition into a stand-alone target table. Until the asynchronous
partition detach operation completes, the target table is unavailable.

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

**Listing 8. Attach partition to main partitioned table**

```
ALTER TABLE FACT_TABLE ATTACH PARTITION STARTING '06-01-2006' ENDING '06-30-2006' FROM TABLE FACT_NEW_MONTH
```
You need to use `SET INTEGRITY` to validate data and maintain indices after the `ATTACH` operation. Starting with DB2 10.1, if data integrity can be checked before attachment, newly attached data can be made available sooner with `SET INTEGRITY IMMEDIATE UNCHECKED`. Otherwise, if `SET INTEGRITY IMMEDIATE CHECKED` is used, commit is required before data is visible.

**Index partitioning**

There are two types of indexing on partitioned tables: partitioned and non-partitioned indices. In general, indices on partitioned tables behave the same as those on non-partitioned tables, but they follow a different storage model:

- While all indices of a non-partitioned table reside in one shared index object within the same table space, non-partitioned indices of a partitioned table are created each in its own index object in a single table space, even if the table partition spans several table spaces.
- A partitioned index uses an index organization scheme in which index data is divided across multiple index partitions, according to the partitioning scheme of the table. Each index partition refers only to table rows in the corresponding data partition. All index partitions for a specific data partition reside in the same index object. By default, index partitions are placed in the same tablespace as the data partitions that they reference. To override this default behavior, you must use the `INDEX IN` clause for each data partition you define by using the `CREATE TABLE` statement. In other words, if you plan to use partitioned indices for a partitioned table, you must anticipate where you want the index partitions to be stored when you create the table. If you try to use `INDEX IN` when creating a partitioned index, you receive an error message.

**Example**

The figure below illustrates the placement of indices on non-partitioned table `table1`. This is done using the `INDEX IN` clause of the `CREATE TABLE` statement.

**Listing 9. Use INDEX IN clause of CREATE TABLE statement**

```sql
CREATE TABLE table1 (a int, b int) IN tbs1 INDEX IN tbs2 CREATE INDEX i1 ON table1 (a) CREATE INDEX i2 ON table1 (b)
```

**Figure 5. Indices on a non-partitioned table**

![Diagram of indices on a non-partitioned table]
Example

The figure below illustrates the placement of partitioned and non-partitioned tables on a partitioned table table1 whose definition is as follows:

```sql
create table table1 (a int, b int) partition by range (a) (starting from (1) ending at (100), partition by range (a) (starting from (101) ending at (200), partition by range (a) (starting from (201) ending at (300)) index in tbs6)
```

Figure 6. Indices on a partitioned table

We have the following indices defined:

- Non-partitioned index i1 resides in table space tbs4: `CREATE INDEX i1 ON table1 (b) NOT PARTITIONED IN tbs4`.
- Non-partitioned index i2 resides in table space tbs5: `CREATE INDEX i2 ON table1 (a,b) NOT PARTITIONED IN tbs5`.
- Partitioned index i3 resides in several table spaces as follows: `CREATE INDEX i2 ON table1 (a) PARTITIONED`. Indices on partitions 1 and 2 reside in the same table spaces as the data partitions they refer to following the default behavior. Data partition 3 overrides the default behavior using the `INDEX IN` clause to specify tbs6 as the tablespace for its partitioned indices.

Note that if the table is partitioned and neither `PARTITIONED` nor `NOT PARTITIONED` is specified, the index is created as partitioned.
Note that the `IN` table space name clause in the `CREATE INDEX` statement can be specified only for a non-partitioned index on a partitioned table. The specification of a table space specifically for the index overrides a specification made using the `INDEX IN` clause when the table was created.

**Range clustered tables**

*Range clustered tables* (RCTs) are tables where data are clustered according to a specific key value. Every key value is associated with its corresponding row location. Hence, this approach provides exceptionally fast access to specific table rows. Key values must have the following characteristics:

- Unique
- Not null
- An integer (SMALLINT, INTEGER, or BIGINT)
- Monotonically increasing
- Within a predetermined set of ranges based on each column in the key

Note that you cannot create a regular index on the same key values used to define the range-clustered table.

DB2 pre-allocates the disk space required for the RCT at creation time. This is done by calculating the number of distinct key values and multiplying it by the table row size. This mandates that the space required for holding the entire table should be available at creation time. Additionally, you cannot issue an `ALTER TABLE` statement to alter the physical characteristics of an RCT after its creation.

RCTs are created using the `ORGANIZE BY KEY SEQUENCE` clause of the `CREATE TABLE` statement as follows:

**Listing 10. Creating RCTs**

```sql
create table table1 (id int not null, emp_id int not null, emp_name varchar(40) ) organize by key sequence
(id starting from 100 ending 1000, emp_id ending 100) disallow overflow
```

This example creates an RCT with the following key characteristics:

- ID values range from 100 to 1,000.
- `emp_id` values range from 1 to 100 (if the `STARTING FROM` clause is not stated, the default starting value is 1).
- The `DISALLOW OVERFLOW` clause indicates that rows with key values outside the specified ranges will not be accepted.

A benefit of overflow-disallowed RCT tables is that table reorganization operations are not required since such tables are always clustered.

In cases you need to allow rows with key values outside the specified range to be stored in the table, you should use the `ALLOW OVERFLOWS` clause while defining the table. In this case, rows with overflow key values are placed in an overflow area, which is dynamically allocated. As more
records are added to this overflow area, operations against the table that involve the overflow area require more processing time. The order of rows in the overflow area is not guaranteed. The larger the overflow area, the more time is required to access it. If this becomes a problem, consider reducing the size of the overflow area by exporting the data to a new RCT with wider ranges.

**Listing 11. Example of an RCT allowing overflows**

```sql
create table table1 (id int not null, emp_id int not null, emp_name varchar(40) ) organize by key sequence (id starting from 100 ending 1000, emp_id ending 100) allow overflow
```

**Multi-dimensional clustering**

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 indices. 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 specified dimension 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 composed 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 indices 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 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:

**Listing 12. Syntax for creating a MDC table**

```
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.

**Figure 7. Data organization within an MDC**

![Diagram of data organization within an MDC]

You cannot alter a table and make it into an MDC table, so use the design adviser, 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 indices.
- 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 indices because each dimension value will need fewer blocks. And, inserts will be quicker because new blocks will be needed less often.
Insert time clustering

Insert time clustering (ITC) is a new partitioning feature coming with DB2 10.1. ITC tables cluster data based on their insertion time so, data inserted within the same time interval are physically grouped together. ITC tables eases the management of space utilization in cases when your system runs batch data deletion and you want to reclaim the trapped free space after the batch deletion is done.

ITC tables have similar characteristics to MDC tables. For example, both table types use block based allocation and block indices. ITC and MDC tables differ in how data is clustered. ITC tables cluster data by using a implicitly created virtual column to cluster rows inserted at a similar time together. ITC virtual dimension cannot be manipulated while clustering dimensions on MDC tables are specified by the creator.

ITC tables are created with the `CREATE TABLE` command by specifying the `ORGANIZE BY INSERT TIME` clause: `CREATE TABLE ... ORGANIZE BY INSERT TIME`. You can convert existing tables to ITC tables using either of two ways:

1. Using the `ADMIN_MOVE_TABLE` procedure to convert the table online.
2. Using the export/import or a load from table utilities.

Note that existing tables cannot be altered to become ITC tables.

Materialized query tables

A materialized query table (MQT) is a table whose definition is based upon the result of a query. The data contained in an MQT is derived from one or more tables accessed by the query on which the MQT definition is based. You can think of an MQT as a kind of materialized view. Views and MQTs are defined on the basis of a query. However, an MQT actually stores the query results as data, and you can work with the data that is in the MQT instead of the data in the underlying tables. Thus, MQTs can significantly improve the performance of queries, especially complex queries.

An MQT can be defined at table creation time as maintained by the system or maintained by the user. When you create this type of MQT, you can specify whether the table data is a REFRESH IMMEDIATE or REFRESH DEFERRED. The REFRESH keyword lets you specify how the data is to be maintained. DEFERRED means that the data in the table can be refreshed at any time using the REFRESH TABLE statement. Neither REFRESH DEFERRED nor REFRESH IMMEDIATE system-maintained MQTs allow insert, update, or delete operations to be executed against them. However, REFRESH IMMEDIATE system-maintained MQTs are updated with changes made to the underlying tables as a result of insert, update, or delete operations.

Example

The following example creates a system maintained MQT with the `REFRESH IMMEDIATE` clause. The `DATA INITIALLY DEFERRED` clause means that data is not inserted into the table as part of the `CREATE TABLE` statement.
Listing 13. Syntax for creating a system maintained MQT

```
CREATE TABLE EMP_DEP AS (SELECT E.EMPNO, E.FIRSTNME, E.LASTNAME, E.PHONENO, D.DEPTNO, SUBSTR(D.DEPTNAME, 1, 12) AS DEPARTMENT, D.MGRNO FROM EMPLOYEE E, DEPARTMENT D WHERE E.WORKDEPT = D.DEPTNO) DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

In the above example:

1. An MQT called EMP_DEP is created joining the data from the EMPLOYEE and DEPARTMENT table.
2. The `DATA INITIALLY DEFERRED` clause means that EMP_DEP MQT is not populated with data as part of the `CREATE TABLE` statement.
3. The `REFRESH IMMEDIATE` clause means that refreshing the data in the MQT is maintained by the system once an update is made to the underlying EMPLOYEE or DEPARTMENT tables.

After being created, the MQT is in check-pending state and cannot be queried until the following `SET INTEGRITY` statement is issued: `SET INTEGRITY FOR EMP_DEP IMMEDIATE CHECKED NOT INCREMENTAL`.

The `IMMEDIATE CHECKED` clause in the above statement specifies that the data is to be checked against the EMP_DEP MQT's defining query and refreshed. The `NOT INCREMENTAL` clause specifies that integrity checking is to be done on the whole MQT and, hence, the MQT definition is recomputed.

**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, 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 9 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.

DB2 10.1 continues adding more cool features to pureXML. New XML indices now more closely match your data, functional indices can speed up searches and queries, binary XML format enables faster data transmission, handling of casting errors is now aligned with SQL, performance
improved for certain XML queries, XML type is now allowed in global variables and also in compiled SQL functions.

**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, 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 data type of XML. Now you can store the XML data in that column.

**XML indices**

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 code below:

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

Starting with DB2 10.1, you can create functional XML indices using the `fn:upper-case` and `fn:exists` functions. Indices created using `fn:upper-case` can speed up case-insensitive searches of XML data. Indices created using `fn:exists` can speed up queries that search for specific elements or for the lack of specific elements. For instance, in our previous example, if you need to speed up case-insensitive searches over the `pname` attribute, you’d use `fn:upper-case` in your index as shown below:

```sql
create index index_name on table_name (xml_column_name) generate key using xmlpattern '/po/purchaser/@pname/fn:upper-case(.)' as sql varchar(50)
```

**New XML index data types**

You can now create indices of type DECIMAL and INTEGER over XML data. In situations where your numeric data is of either INTEGER or DECIMAL type, indices created as DECIMAL and INTEGER values can potentially provide faster query response times. In previous releases, DOUBLE was the only supported numeric type for XML indexes. Consider the following example using an index of type integer.

```sql
CREATE INDEX intidx on favorite_cds(cdinfo) GENERATE KEYS USING XMLPATTERN '/favoritecds/cd/year' AS SQL INTEGER
```

The following SQL data types are supported: VARCHAR, DATE, TIMESTAMP, INTEGER, DECIMAL, and DOUBLE.
New binary XML format

The new binary XML format provides a faster way to transmit and receive XML data between certain Java™ pureXML applications and a DB2 server Version 10.1. For these Java applications, unnecessary XML parsing costs are eliminated, therefore improving performance.

Binary XML data refers to data that is in the Extensible Dynamic Binary XML DB2 Binary XML Format (XDBX). Only Version 4.9 or later releases of the IBM Data Server Driver for JDBC and SQLJ can send or retrieve XML data to/from the data server in XDBX format. You have the full control over the XML format you use for transmission using the xmlFormat property either on the data source level or the connection properties.

Binary XML format is most efficient for cases in which the input or output data is in a non-textual representation, such as SAX, StAX, or DOM. For example, these methods retrieve XML data in non-textual representations.

Table compression

You can use less disk space for your tables by taking advantage of the DB2 table compression capabilities. Compression saves disk storage space by using fewer database pages to store data. You can use compression with new and existing tables. Temporary tables are also compressed automatically, if the database manager deems it to be advantageous to do so. Row compression uses a dictionary-based compression algorithm to replace recurring strings with shorter symbols within data rows.

Table row compression types

There are two types of row compression you can choose from:

- Classic row compression
- Adaptive compression

Data stored within data rows, including inlined LOB or XML values, can be compressed with adaptive and classic row compression. XML storage objects can be compressed using static compression. However, storage objects for long data objects stored outside table rows is not compressed.

Classic row compression

Classic row compression, sometimes referred to as static compression, uses a table-level compression dictionary to compress data by row. The dictionary is used to map repeated byte patterns from table rows to much smaller symbols; these symbols then replace the longer byte patterns in the table rows. The compression dictionary is stored with the table data rows in the data object portions of the table.

Using row compression

To use row compression, you must first set the table to be compression eligible, 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 static or alter table table_name compress yes static.

Creating the row 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 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 to rebuild the compression dictionary) you must run the command reorg table table_name resetdictionary.

This will scan the table, create the dictionary, and 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 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.

To disable compression for a table, use the ALTER TABLE statement with the COMPRESS NO option; rows you later add are not compressed. Existing rows remain compressed. To extract the entire table after you turn off compression, you must perform table reorganization with the REORG TABLE command:

alter table tablename compress no reorg table table_name

Adaptive compression

Adaptive compression actually uses two compression approaches. The first employs the same table-level compression dictionary used in classic row compression to compress data based on repetition within a sampling of data from the table as a whole. The second approach uses a page-level dictionary-based compression algorithm to compress data based on data repetition within each page of data. The dictionaries map repeated byte patterns to much smaller symbols; these symbols then replace the longer byte patterns in the table. The table-level compression dictionary is stored within the table object for which it is created and is used to compress data throughout the table. The page-level compression dictionary is stored with the data in the data page and is used to compression only the data within that page.

Using adaptive compression

You compress table data using adaptive compression by setting the COMPRESS attribute of the table to YES ADAPTIVE or to YES, which by default enables adaptive compression. You can set this attribute when you create the table by running the command create table tablename .. compress yes adaptive or create_table tablename .. compress yes.
You can also alter an existing table to use compression by using the same options for the ALTER TABLE statement:

```
alter table tablename compress yes adaptive
```

HINT: If you have scripts or applications that issue the ALTER TABLE or CREATE TABLE statements with the COMPRESS YES clause, make sure you add the STATIC or ADAPTIVE keyword to explicitly indicate the table compression method you want.

After you enable compression, operations that add data to the table, such as an INSERT, LOAD INSERT, or IMPORT INSERT command operation, can use adaptive compression. In addition, index compression is enabled for the table. Indices are created as compressed indices unless you specify otherwise and if they are the types that can be compressed.

To disable compression for a table, use the ALTER TABLE statement with the COMPRESS NO option; rows that you later add are not compressed. Existing rows remain compressed. To extract the entire table after you turn off compression, you must perform a table reorganization with the REORG TABLE command.

**Compress temp tables**

Compression for temporary tables is enabled automatically with the DB2 Storage Optimization Feature. Only classic row compression is used for temporary tables. When executing queries, the DB2 optimizer considers the storage savings and the impact on query performance that compression of temporary tables offers to determine whether it is worthwhile to use compression. If it is worthwhile, compression is used automatically.

**Estimating space savings**

The ADMIN_GET_TAB_COMPRESS_INFO table function estimates the compression savings that can be gained for the table, assuming a REORG with RESETDICTIONARY option will be performed. The following example uses the ADMIN_GET_TAB_COMPRESS_INFO table function to estimate percentage saved for classic and adaptive compression if enabled on the DB2INST1.CUSTOMERS table:

```
SELECT TABNAME, OBJECT_TYPE, ROWCOMPMODE, PCTPAGESSAVED_CURRENT current, PCTPAGESSAVED_STATIC with_static, PCTPAGESSAVED_ADAPTIVE with_adaptive FROM TABLE(SYSProc.ADMIN_GET_TAB_COMPRESS_INFO('DB2INST1','CUSTOMERS')) AS T
```

**Figure 8. Result for the example using ADMIN_GET_TAB_COMPRESS_INFO table function**

<table>
<thead>
<tr>
<th>TABNAME</th>
<th>OBJECT_TYPE</th>
<th>ROWCOMPMODE</th>
<th>CURRENT</th>
<th>WITH_STATIC</th>
<th>WITH_ADAPTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMERS</td>
<td>DATA</td>
<td>S</td>
<td>60</td>
<td>60</td>
<td>01</td>
</tr>
<tr>
<td>CUSTOMERS</td>
<td>XNL</td>
<td>S</td>
<td>58</td>
<td>62</td>
<td>62</td>
</tr>
</tbody>
</table>

**Temporal tables**

**Introduction to temporal tables**

A new feature included in DB2 10.1 is the time-travel or temporal data tables. Temporal tables allow you track the changes in your data so that you can back in time and query the data at a specific point in time. You can also control the business validity of your data. Temporal data can
be used mainly in two flavors: system-period and application-period. You can combine the two to create a third: the so-called bi-temporal tables. Using new and standardized SQL syntax, you can easily insert, update, delete, and query data in the past, present, or future.

The temporal data management capabilities in DB2 are based on three types of temporal tables. The type of temporal table you are going to use is controlled by the nature of the time fields it’s going to manage. Is it a system time? Business time? Or both? System time is maintained by DB2 system and is provided by the system clock on DB2 server when a transaction occurs. Business time is date or time stamp provided by the application to control the business validity of records. It’s completely unrelated to the system time on DB2 server.

**System-period temporal tables (STT)**

*System-period temporal tables* (STT) are based on the system time. DB2 keeps a history of rows that have been updated or deleted over time according to system time. Each row and its corresponding history are automatically assigned a pair of system time *(transaction time)*.

**Application-period temporal tables**

Unlike STT, *application-period temporal tables* (ATT) require that the application is aware of the ATT to provide a valid business time. Applications control the business validity of their data by supplying a date or timestamp. The provided dates or timestamps describe when the data in a given row was or will be valid in the real world. Again, new SQL constructs allow users to insert, query, update, and delete data in the past, present, or future. DB2 automatically applies temporal constraints and row-splits to correctly maintain the application-supplied business time.

**Bi-temporal tables**

*Bi-temporal tables* manage system and business time, and combine all the capabilities of both. This enables applications to manage the business validity while DB2 keeps a full history of data changes. Every bi-temporal table is both an STT and an ATT.

One key difference between STT and ATT is that an STT has an associated separate history table to maintain previous versions of the data. An ATT has no separate history table; all rows are maintained within the same table. When rows from ATT are affected by delete operations, they are permanently deleted from the table.

Table 4 gives a quick overview of the characteristics of system time and business time. If your requirements match the characteristics of system time, you can use STT. If they match the characteristics of business time, go for ATT. If your application needs both characteristics, you can leverage it to the bi-temporal table.

**Table 4. Characteristics of system time and business time**

<table>
<thead>
<tr>
<th></th>
<th>System time</th>
<th>Business time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Captures the time</td>
<td>Captures the time when changes happen to data inside a DB2 database</td>
<td>Captures the time when changes happen to business objects in the real world</td>
</tr>
<tr>
<td>when changes happen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to data inside a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB2 database</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consider the following guidelines when you choose temporal tables:

- Use system time and system-period temporal tables to track and maintain a history of when information was physically inserted, updated, or deleted inside your DB2 database.
- Use business time and application-period temporal tables, if you need to describe when information is valid in the real world, outside of DB2.
- Use bi-temporal tables, if you need to track both dimensions of time. With a bi-temporal table you can manage business time with full traceability of data changes and corrections.

**System-period temporal tables**

**Creating a system-period temporal table**

To create a system-period temporal table that leverages system time and versioning, you need to create a base table with three specific columns, create an identical history table for historical data, and finally associate the history table with the base table. Let's look at these three steps in more details:

1. **Create a base table with three specific columns**
   The three specific columns are row-begin column, row-end column, and transaction start-ID column. The names of three columns can be any valid DB2 column name (e.g., start_time, end_time, and trans_id). Each column must be defined as a TIMESTAMP. The row-begin and row-end column names must be specified in the PERIOD SYSTEM_TIME (start_time, end_time) clause.

   **Listing 14. Sample DDL table with three specific columns**

   ```sql
   CREATE TABLE travel( trip_name CHAR(30) NOT NULL PRIMARY KEY, destination CHAR(12) NOT NULL, departure_date DATE NOT NULL, price DECIMAL (8,2) NOT NULL, sys_start TIMESTAMP(12) NOT NULL generated always as row begin implicitly hidden, sys_end TIMESTAMP(12) NOT NULL generated always as row end implicitly hidden, tx_start TIMESTAMP(12) generated always as transaction start id implicitly hidden, PERIOD SYSTEM_TIME (sys_start, sys_end) );
   ```

2. **Create an identical history table for historical data**
   The history table is logically identical to the base table. However, it can be physically different from the base table. You can put it in a different table space, or even you can have different partitioning schemes for each table. You may use the like clause in the create table statement to create a history table with the same names and descriptions as the columns of the base table as follows: `CREATE TABLE travel_history LIKE travel IN hist_space WITH RESTRICT ON DROP;`.

3. **Associate the history table with the base table**
The last step in setting up your system-period temporal table is to link the history table with the base table. This can be easily done with the `ALTER TABLE` command:

```
ALTER TABLE travel
ADD VERSIONING USE HISTORY TABLE travel_history;
```

You can hide the special columns by specifying the `IMPLICITLY HIDDEN` clause in the column definition. In that case, if you don't use the like clause when creating the history table, you must mark all hidden columns in the base as hidden in the history table as well.

You can also change your existing table to leverage the temporal table using the same steps. In that case, the base table already exists without the three columns. Alter the table adding these columns and adding the `PERIOD SYSTEM_TIME(sys_start, sys_end)`.

**DML operations against System-period temporal table**

**Application-period temporal tables**

`ATT` allows you to store business time. By "business time," we mean the application logical notion of time. That is, it reflects when information was, is, or will be effective (valid) in the real world. Business time by itself does not involve a history table, and the effective dates must be provided by the application. We are talking about the validity of the data not the history of the data.

Let's take an example to make it clear by an example. Consider the following two very different questions:

- What was the effective price for Manu Wilderness trip on 06/15/2012? *(Business time)*
- What was the stored price for Manu Wilderness trip on 06/15/2012? *(System time)*

For example, if a trip will be offered at a discounted price during June, the price change might get entered into the database today or at anytime (system time) with an effective date from 1 Jun 2012 to 1 Jul 2012 (business time). Similarly, a new insurance policy might get created and inserted into the database in May, but backdated with an effective start date of 15 Apr. In short, the business validity for a record is independent of when that record was or was not physically present in the database.

**Creating application-period temporal tables**

Similar to STT, you will need a pair of DATE or TIMESTAMP columns that describe the start and end points of the business validity of each record. Although these columns can have arbitrary names, they must be declared using the `BUSINESS_TIME` period clause in the `CREATE TABLE` statement. This period declaration enables DB2 to enforce temporal constraints and to perform temporal `UPDATE` and `DELETE` operations for business time. DB2 will transparently add, split, or delete rows as needed.

Let's create a sample, run the following DDL.

**Listing 15. Sample DDL to create application-period temporal table**

```
CREATE TABLE travel_att ( trip_name CHAR(25) NOT NULL, destination CHAR(8) NOT NULL, price DECIMAL(8,2) NOT NULL, bus_start DATE NOT NULL, bus_end DATE NOT NULL, PERIOD BUSINESS_TIME (bus_start, bus_end), PRIMARY KEY (trip_name, BUSINESS_TIME WITHOUT OVERLAPS) );
```
The optional clause `BUSINESS_TIME WITHOUT OVERLAPS` in the primary key constraint is new in DB2 10.1 and enforces temporal uniqueness. Temporal uniqueness means, rows with duplicate `trip_name` values are allowed if the business time periods of these rows do not overlap. In other words, `BUSINESS_TIME WITHOUT OVERLAPS` means that for any given point in time, there is at most one row that is effective (valid) for a given `trip_name`.

**Insert data into application-period temporal tables**

Inserting new record into an ATT is no different from the regular SQL insert statement. You simply need to supply appropriate values for all required columns, including the columns representing business time start and end values. For example, to insert two rows into our sample travel table with business time, we could issue the following statement:

```
INSERT INTO travel_att VALUES ('Manu Wilderness', 'Peru', 1500.00, '05/01/2012', '01/01/2013'), ('The Great Wall Tour', 'China', 2200.00, '07/01/2012', '12/30/9999');
```

At this point, the data in the `travel_att` table should look like below:

<table>
<thead>
<tr>
<th>TRIP_NAME</th>
<th>DESTINATION</th>
<th>PRICE</th>
<th>BUS_START</th>
<th>BUS_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manu Wilderness</td>
<td>Peru</td>
<td>1500.00</td>
<td>2012-05-01</td>
<td>2015-10-01</td>
</tr>
<tr>
<td>The Great Wall Tour</td>
<td>China</td>
<td>2200.00</td>
<td>2012-07-01</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

As you can see, the `TRAVEL_ATT` table doesn't contain any information about when the rows were inserted, updated, or deleted. It only tracks validity (business time) of the trips.

**Update data in an application-period temporal table**

The company discovered that the demand on Manu Wilderness trip is very low in July. They made an offer for July reducing its price to $1,000 instead of $1,500. This offer is only valid for July. The price remains the same for the remaining period until the trip becomes invalid:

```
UPDATE travel_att FOR PORTION OF BUSINESS_TIME FROM '06/01/2012' TO '07/01/2012' SET price = 1000.00 WHERE trip_name = 'Manu Wilderness';
```

The command completed successfully and the period from 5 May 2012 to 1 Oct 2015 is split into three periods. DB2 inserted two new rows and updated the existing row as shown below:

```
Listing 16. DB2 inserts two new rows and updates existing row

<table>
<thead>
<tr>
<th>TRIP_NAME</th>
<th>DESTINATION</th>
<th>PRICE</th>
<th>BUS_START</th>
<th>BUS_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manu Wilderness</td>
<td>Peru</td>
<td>1000.00</td>
<td>2012-06-01</td>
<td>2012-07-01</td>
</tr>
<tr>
<td>The Great Wall Tour</td>
<td>China</td>
<td>2200.00</td>
<td>2012-07-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>Manu Wilderness</td>
<td>Peru</td>
<td>1500.00</td>
<td>2012-05-01</td>
<td>2015-10-01</td>
</tr>
</tbody>
</table>
```

Now if we try to insert a record with a period overlapping with the existing period, DB2 will restrict you and the command will fail. For example, if you try to insert a new record with trip name Manu Wilderness and an overlapping period, DB2 will fail with SQL error -0803. In that case, you may change the name or update the existing record based on your business needs.

**Delete data from an application-period temporal table**

Deleting records from an application-period temporal table removes rows from the table and can potentially result in new rows inserted into the application-period temporal table itself. A row is a
candidate for deletion if its period-begin column, period-end column, or both fall within the range specified in the FOR PORTION OF BUSINESS_TIME clause. If a specified period falls in the middle of an existing period, DB2 will transparently split the exiting period into three periods and delete the correct one. Let’s clarify it more with the following example:

```
DELETE FROM travel_att FOR PORTION OF BUSINESS_TIME FROM '09/15/2012' TO '09/20/2012' WHERE trip_name LIKE 'Manu Wilderness';
```

When you delete for a portion of business time, you are not necessarily removing records from the database. You are marking this period as invalid for the application. You are making it logically unavailable from the business perspective. However, this may not be implemented by a physical delete. In our example, the Manu Wilderness is available in the period from 1 Jul 2012 to 1 Oct 2015. Marking this trip is not available (delete for portion of business time) in the period from 15 Sep 2012 to 20 Sep 2012 means that there will be a gap. To keep the records before and after this gap, the current record will be split into two rows. In other words, the existing record will be updated and a new record will be inserted:

**Listing 17. Existing record updated and new record inserted**

```
TRIP_NAME DESTINATION PRICE BUS_START BUS_END
------------ ----------- ---------- -----------
----------- Manu Wilderness Peru 1000.00 2012-06-01 2012-07-01
9999-12-30 Manu Wilderness Peru 1500.00 2012-05-01 2012-06-01
2012-09-15 Manu Wilderness Peru 1500.00 2012-09-20 2015-10-01
```

DB2 manages all system time and business time periods as inclusive-exclusive periods, or closed-open periods. Therefore, it is recommended to also use inclusive-exclusive periods at the application level and to avoid mapping between inclusive-inclusive and inclusive-exclusive periods.

**Bi-temporal tables**

Bi-temporal tables combine all the capabilities and restrictions that apply on system-period and application-period temporal tables. You need two columns for the systems time period and another two for the business time period, as well as the transaction start-ID column.

**Creating bi-temporal tables**

Administrators can easily create or alter a table to include system and business time. For example, the following `CREATE TABLE` statement defines a bi-temporal table with a BUSINESS_TIME period on the BUS_START and BUS_END columns, as well as a SYSTEM_TIME period on the SYS_START and SYS_END columns.

**Listing 18. Sample DDL to create bi-temporal tables**

```
CREATE TABLE policy ( id INT NOT NULL, vin VARCHAR(10), annual_mileage INT, rental_car CHAR(1), coverage_amt INT, bus_start DATE NOT NULL, bus_end DATE NOT NULL, sys_start TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL, sys_end TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL, trans_start TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN, PERIOD SYSTEM_TIME (sys_start, sys_end), PERIOD BUSINESS_TIME(bus_start, bus_end), PRIMARY KEY(id, BUSINESS_TIME WITHOUT OVERLAPS) );
```
Querying temporal tables

Temporal tables special registers

CURRENT TEMPORAL BUSINESS_TIME (for ATT) and CURRENT TEMPORAL SYSTEM_TIME (for STT) can be used to change the current time for a particular query session. This allows database queries as of a point in time in the past or future, even if the packaged application does not allow its SQL statement syntax to be modified.

The new SQL constructs allow you to query the system-period temporal table or views on an STT to return results for a specified period or point in time. Querying temporal tables can be summarized in five basic examples:

1. Query without time specification — This leverages the temporal tables special registers if they are set to a not null value. Otherwise, it retrieves the current data.
2. Query with AS OF value clause — Includes all the rows where the begin value for the period is less than or equal to value1 and the end value for the period is greater than value1. This enables you to query your data as of a certain point in time.
3. Query with FROM value1 TO value2 clause — Includes all the rows where the begin value for the period is equal to or greater than value1 and the end value for the period is less than value2. This means that the begin time is included in the period, but the end time is not.
4. Query with BETWEEN value1 AND value2 clause — Includes all the rows where any time period overlaps any point in time between value1 and value2. A row is returned if the begin value for the period is less than or equal to value2 and the end value for the period is greater than value1.
5. Query views defined on a temporal table — You can either set the time specification at the query or in the view DDL. However, you can’t have time specified on both.

Partitioning temporal tables

Any temporal table can also be a range-partitioned table. A history table that belongs to a system period temporal table or a bi-temporal table can also be range-partitioned. The partitioning scheme of a history table may differ from the partitioning of its base table. For example, you can partition the current data by month and the history data by year. However, ensure that the ranges you defined for the history table partitions can absorb any rows moved from the base table to the history table. Consider the following example.

Listing 19. Example for table partitioning

```
CREATE TABLE policy ( id INTEGER PRIMARY KEY NOT NULL, annual_mileage INTEGER, rental_car CHAR(1), coverage_amt INTEGER, sys_start TIMESTAMP(12) GENERATED AS ROW BEGIN NOT NULL, sys_end TIMESTAMP(12) GENERATED AS ROW END NOT NULL, trans_start TIMESTAMP(12) GENERATED AS TRANSACTION START ID, PERIOD SYSTEM_TIME (sys_start, sys_end) ) PARTITION BY RANGE(sys_start) (STARTING('2012-01-01') ENDING ('2014-12-31') EVERY 1 MONTH ); CREATE TABLE policy_history ( id INTEGER NOT NULL, annual_mileage INTEGER, rental_car CHAR(1), coverage_amt INTEGER, sys_start TIMESTAMP(12) NOT NULL, sys_end TIMESTAMP(12) NOT NULL, trans_start TIMESTAMP(12)) PARTITION BY RANGE(sys_start) (STARTING('2012-01-01') ENDING ('2013-12-31') EVERY 3 MONTHS ); ALTER TABLE policy ADD VERSIONING USE HISTORY TABLE policy_history;
```

The example shows different partitioning schemes for the base and history table. The policy table is partitioned by month; the policy_history table is partitioned by quarter. This difference
in granularity is not a problem. However, the policy table has partitions for sys_start values up to 2014-12-31 while the policy_history table has partitions only up to 2013-12-31. If a row in the policy table with a sys_start value of 2014-01-01 or greater is deleted, the insert into the history table fails because it has no partition to accept this sys_start value. Hence, the entire delete transaction fails. To avoid this, define the history table such that the total range of all its partitions is always equal to or greater than the total range of partitions in the base table.

After versioning has been enabled with the \texttt{ALTER TABLE...ADD VERSIONING} statement, you can still detach partitions from the history table for pruning and archiving purposes. However, to detach a partition from the base table, you must first stop versioning with the \texttt{ALTER TABLE...DROP VERSIONING} statement.

When you stop versioning and detach a partition from the base table, this partition becomes an independent table. It retains all three timestamp columns (row begin, row end, transaction start ID) but not the \texttt{PERIOD SYSTEM_TIME} declaration. The rows in the detached partition are not automatically moved into the history table. If you choose to move these rows into the history table yourself, change the sys_end value of every row from 9999-12-30-00.00.00.000000000000 to the current timestamp. This change is necessary to reflect the point in time when the rows changed from being current to being history. If you do not make this change, temporal queries might return unexpected results.

You can attach a table to a partitioned base or history table while versioning is enabled. The table you attach is not required to have a \texttt{PERIOD SYSTEM_TIME} declaration, but it must have all three timestamp columns defined as in the base table. While versioning is enabled, you cannot use the \texttt{SET INTEGRITY} statement with the \texttt{FOR EXCEPTION} clause. The reason is that moving any exception rows into an exception table cannot be recorded in the history table, which jeopardizes the ability to audit of the base table and its history. However, you can temporarily disable versioning, perform \texttt{SET INTEGRITY} with the \texttt{FOR EXCEPTION} clause, then enable versioning again.

\textbf{ADMIN\_MOVE\_TABLE procedure and temporal tables}

There are some limitations when using the \texttt{ADMIN\_MOVE\_TABLE} stored procedure to move data in an active system-period temporal table into a new table with the same name. The following actions are blocked:

- Alter table operations that change the definition of the system-period temporal table or the associated history table are blocked during online move operations.
- The KEEP option of \texttt{ADMIN\_MOVE\_TABLE} is unavailable for system-period temporal tables.

Additionally, the online-table-move operation is not supported for history tables.

\textbf{Multi-temperature data feature}

You can configure your databases so that frequently accessed data (\textit{hot data}) is stored on fast storage, infrequently accessed data (\textit{warm data}) is stored on slightly slower storage, and rarely accessed data (\textit{cold data}) is stored on slow, less-expensive storage. As hot data cools down and is accessed less frequently, you can dynamically move it to the slower storage.
Multi-temperature storage provides the ability to assign priority to data (hot, warm, cool, cold) and dynamically assign it to different classes of storage. This is carried out by the introduction of storage groups, which is a new layer of abstraction between logical table spaces and physical storage (containers).

Using storage groups to prioritize your data

Storage groups allow the flexibility to implement multi-temperature data management in automatic storage table spaces. After you create storage groups that map to the different classes of storage in your database management system, you can assign automatic storage table spaces to those storage groups, based on which table spaces have hot, warm, or cold data.

The following example illustrates the use of storage groups with multi-temperature data. As shown in the figure, solid-state drives (SSD) are used to hold data for the current quarter, and enough Fibre Channel-based (FC) and Serial Attached SCSI (SAS) drives are used to hold data for the previous three quarters. The data that is older than one year is stored on a large Serial ATA (SATA) RAID array that does not perform quickly enough to withstand a heavy query workload.

Figure 9. Usage of storage groups with multi-temperature data

Changing the temperature of your data

You can dynamically reassign a table space to a different storage group as the data changes or your business direction changes. The ALTER TABLESPACE statement can be used to do this: ALTER TABLESPACE tbSpc USING STOGROUP sg_target.

After the ALTER TABLESPACE statement is committed, containers are allocated on the new storage group's storage paths, the existing containers residing in the old storage groups are marked as drop pending, and an implicit REBALANCE operation is initiated. This operation allocates containers on the new storage path and rebalances the data from the existing containers into the new containers. The number and size of the containers to create depend on both the number of storage paths in the target storage group and on the amount of free space on the new storage paths. The old containers are dropped, after all the data is moved.
Multi-temperature storage integrates with DB2 Workload Manager

DB2 Workload Manager (WLM) gives users the ability to prioritize incoming work based on what data is accessed (a data-centric approach). Users can assign a data tag attribute (a value from 0 to 9) to a storage group or tablespace. The data tag can be used by WLM to determine how to treat the work.

WLM can use data tags predictively. The Optimizer gathers a list of data tags for all table spaces used by an SQL statement at compile time. The data tag can influence the initial placement of the statement into a service class. WLM can also use data tags reactively. At runtime, based on the tag of a table space being accessed, a statement can be remapped into a lower-priority subclass.

A sample scenario

The following steps provide more details on how to set up multi-temperature data storage for the example shown above:

1. Create three storage groups to reflect the three classes of storage, a storage group to store hot data, a storage group to store warm data, and a storage group to store cold data.

   ```sql
   CREATE STOGROUP sg_hot ON '/hot/fs1' DATA TAG 1
   CREATE STOGROUP sg_warm ON '/warm/fs1', '/warm/fs2' DATA TAG 5
   CREATE STOGROUP sg_cold ON '/cold/fs1', '/cold/fs2', '/cold/fs3' DATA TAG 9
   ```

   Data tags values assigned to storage groups sg_hot and sg_warm to indicate the business priority of the data stored in these storage groups to the WLM.

2. Create four table spaces, one per quarter of data in the year, and assign the table spaces to the storage groups. Data tags represent business priority to be used by the optimizer.

   ```sql
   CREATE TABLESPACE tbsp_2012q1 USING STOGROUP sg_warm
   CREATE TABLESPACE tbsp_2012q2 USING STOGROUP sg_warm
   CREATE TABLESPACE tbsp_2012q3 USING STOGROUP sg_warm DATA TAG 3
   CREATE TABLESPACE tbsp_2012q4 USING STOGROUP sg_hot
   ```

   This association results in table spaces inheriting the storage group properties. Table spaces tbsp_2012q1 and tbsp_2012q2 will inherit the data tag value of 5 from the sg_warm storage group, but table space tbsp_2012q3 will not inherit the data tag value from the storage group because another data tag value has been assigned to the table space. Table space for data tbsp_2012q4 will inherit the data tag value of 1 from the sh_hot storage group.

3. Set up your range partitions in your table:

   ```sql
   LISTING 20. Set up range partitions
   CREATE TABLE ... PARTITION BY RANGE (date_column) (PART "2012Q1" STARTING ('2012-01-01') ENDING ('2012-03-31') in "tbsp_2012q1", PART "2012Q2" STARTING ('2012-04-01') ENDING ('2012-06-30') in "tbsp_2012q2", PART "2012Q3" STARTING ('2012-07-01') ENDING ('2012-09-30') in "tbsp_2012q3", PART "2012Q4" STARTING ('2012-10-01') ENDING ('2012-12-31') in "tbsp_2012q4"
   ```

   The 2012Q4 data represents the current quarter and is using the sg_hot storage group.

4. After the current quarter passes, create a table space for the new quarter and assign the table space to the sg_hot storage group: CREATE TABLESPACE tbsp_2013q1 USING STOGROUP sg_hot.

5. Move the table space for the quarter that just passed to the sg_warm storage group. To change the storage group association for the tbsp_2012q4 table space, issue the ALTER
TABLESPACE statement with the USING STOGROUP option. To fix the business priority for the altered tablespace, the DATA TAG option will be used. Additionally, table space tbsp_2012q3 will be altered to change the data tag value from 3 to 5:

```sql
ALTER TABLESPACE tbsp_2012q4 USING STOGROUP sg_warm DATA TAG 3
ALTER TABLESPACE tbsp_2012q3 DATA TAG 5
```

6. Finally, move the table space for the older quarter assigned to sg_warm storage group to sg_cold storage group. To change the storage group association for the tbsp_2012q1 table space, issue the ALTER TABLESPACE statement with the USING STOGROUP option:

```sql
ALTER TABLESPACE tbsp_2012q1 USING STOGROUP sg_cold.
```

**Conclusion**

This tutorial focuses on the physical database design concepts and elements. You should now be able to do the following:

- Demonstrate the ability to create a database and manipulate various DB2 objects
  - Know how to convert existing database to automatic storage database
  - Use the Admin_move_table feature
- Demonstrate knowledge of partitioning capabilities
- Demonstrate knowledge of XML data objects
- Demonstrate knowledge of compression
- Demonstrate knowledge of new table features
- Knowledge of Multi-temperature data feature
Related topics

• Read the Preparation Guide for DB2 10.1 DBA for Linux, UNIX, and Windows Exam 611 to learn in-depth information about each of the concepts presented in this tutorial. This guide is compilation of topics from the DB2 10.1 documentation.
• Use the DB2 documentation in IBM Knowledge Center to find more details about each of the concepts presented in this tutorial.
• Use an RSS feed to request notification for the upcoming tutorials in this series. (Find out more about RSS feeds of developerWorks content.)
• Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.

© Copyright IBM Corporation 2013
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)