

DB2 9 DBA exam 731 prep, Part 3: Database access

George Baklarz

July 18, 2006

This tutorial will take you through the various steps required to manage some of the objects within DB2®. It includes overview information for indexes, constraints, referential integrity, and views. This is the third in a [series of seven tutorials](#) that you can use to help prepare for the DB2 9 for Linux®, UNIX®, and Windows™ Database Administration Certification (Exam 731).

[View more content in this series](#)

Before you start

About this series

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

About this tutorial

This tutorial will take you through the various steps required to manage some of the objects within DB2. The following topics are included:

- Creating DB2 tasks using the GUI tools
- Creating and managing indexes
- Creating constraints on tables (for example, RI, informational, unique)
- Creating views on tables
- Examining the contents of the system catalog tables
- Enforcing data uniqueness

This is the third tutorial in a series of seven to help you prepare for the DB2 9 for Linux, UNIX®, and Windows™ Database Administration Certification (Exam 731). The material in this tutorial primarily covers the objectives in Section 3 of the exam, entitled "Database Access." You can view these objectives at: <http://www-03.ibm.com/certify/tests/obj731.shtml>.

DB2 installation is not covered in this tutorial. If you haven't already done so, we strongly recommend that you download and install a copy of [IBM DB2 9](#). Installing DB2 will help you understand many of the concepts that are tested on the DB2 9 Database Administration Certification exam.

Prerequisites

Since you are reading this tutorial, you obviously want to be educated in the nuances of administering DB2, or perhaps to upgrade your current skills to reflect the new features found in the most recent release of the product. You could also be interested in getting certified in the administration of DB2 and need to review what we cover in the exam!

Whatever your reason, there are a few things you should do to get the most from this and other tutorials:

1. Get a copy of DB2 9 . Without something to practice on, you won't be able to try out the examples or explore the various features of the product.
2. Take the [DB2 9 Fundamentals exam prep tutorials](#). These tutorials give you the fundamentals on DB2 and will make it easier to understand some of the terminology that is used in this tutorial. It probably wouldn't be a bad idea if you passed that [certification test](#) before trying this one!
3. Practice and try things out on your own. While this tutorial tries to cover many of the topics you need to get certified, nothing beats experience.
4. Read the DB2 Administration manuals. They can give you a lot of insight on how DB2 works. There are also other sources of information on DB2 that can be found at the end of this tutorial (see [Resources](#)).

Create DB2 tasks using the GUI tools

DB2 Task Center

One of the DB2 tools that is used to control the database is the *Task Center*. The Task Center is used to run tasks, either immediately or according to a schedule, and to notify people about the status of completed tasks.

The Task Center includes all the functionality found in the Script Center in previous versions of DB2, plus additional new features. A *task* is a script, together with associated success conditions, schedules, and notifications. You can create a task within the Task Center, create a script within another tool and save it to the Task Center, import an existing script, or save the options from a DB2 dialog or wizard such as the Load wizard. A script can contain DB2, SQL, or operating system commands.

Create the tools database

To use the Task Center, the tools catalog must exist. The tools catalog contains information about the administrative tasks that you configure with such tools as the Task Center and Control Center.

The tools catalog may have been created for you when you installed DB2, but in the event that it is not available, you can create it using either a DB2 command or the tools menu.

To create the tools database from the command line, open up a DB2 command window and issue the following command:

```
db2 create tools catalog cc create new database toolsdb
```

To create the database using the tools settings, select **Tools > Tools Settings > Scheduler Settings** from within the Control Center:

Start the Task Center

Start the Task Center with the command `db2tc` from a command line, or click the Task Center from any of the DB2 tools:

In addition, the Task Center can be started from the DB2 menu in a Windows environment:

Once the Task Center is started, you can manage, create, and run scripts.

Task Center functionality

For each task, you can do the following:

- Schedule the task
- Specify success and failure conditions
- Specify actions that should be performed when this task completes successfully or when it fails
- Specify e-mail addresses (including pager addresses) that should be notified when this task completes successfully or when it fails

You can specify conditional coding by creating *task actions*. Each task action consists of a task and the action that should be performed on the task. For example, task 1 can have the following task actions:

- If task 1 is successful, task action A enables the scheduling of task 2.
- If task 1 fails, task action B runs task 3.

You can also create a *grouping task* that combines several tasks in a single logical unit of work. When the grouping task meets the success or failure conditions that you define, any follow-on tasks are run. For example, you can combine three backup scripts into a grouping task and then specify a reorganization as a follow-on task that will be executed if all of the backup scripts execute successfully.

Create a task

The Task Center can require up to eight steps to generate a task:

1. Describe and name the task
2. Select a command script
3. Define the success or failure criteria (run properties)
4. Group tasks
5. Schedule the task to run
6. Set a notification level
7. Select task actions after completion
8. Set security

Each of these steps is described in the following panels. When the Task Center is started, it displays all of the tasks that are currently defined in the system:

To create a new task, either left-click in the task listing area, or select **Task > New** from the menu at the top of the panel.

This opens a window that allows you to define the task you want to create.

The Task Center is updated with any new tasks that you create. Note that the refresh options should be set to a reasonable time interval if you want to see any changes to the tasks that are already in the list.

Describe and name the task


The task description panel lets you define the characteristics of the task itself:

The Name field contains the name of the task that you are creating. This name can contain any character, so a descriptive name can be used.

The Type field tells DB2 the type of action that this task is going to perform. There are four options:

- **DB2 command script:** This script issues DB2 commands, which can contain SQL.
- **OS command script:** Operating system commands are included in this type of script.
- **MVS shell script:** MVS shell scripts execute in an MVS or z/OS host environment. This type of task can also include JCL (Job Control Language) statements.
- **Grouping task:** Grouping task takes several tasks and places them into one group that is executed together.

The Description field can contain a lengthy description of the task being created.

The Task Category field is used to classify the type of task that is being created. This is an optional field, but can be very useful when you're trying to find a previous command, or to group commands together based on their purpose. Click the ellipsis button at the end of the field  to see a list of defined categories:

A task can have multiple categories associated with it. The available task categories can be used to classify the task by selecting the category and then clicking the **>** button. Additional categories can be created by typing a new entry under **New task category** and clicking **>**. These categories can be used to sort the available tasks in the main task window at a later time.

The Run System field indicates the system on which the command will run. This can be a local system (as is the case in the sample figures) or any remote system that has been cataloged.

The DB2 Instance and Partition option lets you select the instance and partition on which the command will run. The partition option is meant only for systems that have the DPF (DB2 Partitioning Feature) installed. In such a case, the task can run on specific partitions within the database, rather than across all of them.

All of the fields, except for the description and categorization, need to be filled in for the task to be created.

Select a command script

The *command script* panel contains the actual DB2 or operating system command that you want to run against the database. The main window lets you type in the command or use cut and paste operations to place the information into this panel.

If you have created or saved a script in an external file, click the Import button to load it.

If a script needs to access data or output information to disk, specify the default directory so that DB2 knows where to place or find information.

Finally, the DB2 statement terminator refers to the character that will be used to delimit multiple DB2 statements. By default, this is the semicolon (;), but you may need to change this if you are creating DB2 functions, triggers, or any SQL PL statements. These DB2 objects use semicolons as their own statement delimiter, so a different DB2 delimiter should be used (dollar signs (\$) or at signs (@) are both popular symbols to use).

Run properties

The Run Properties panel is used to define the success or failure criteria of the task. You can either tell the script to stop when there is any non-zero return code (stop execution at first return code that is a failure) or define a set of return code conditions.

If a task can return multiple return codes that are considered successful, you need to define a success code set. To display a list of code sets, click the Success code set field.

In the event that there are no success code sets displayed, or if none of them match your requirements, create your own by clicking **New**.

You can create your own success code set, which can contain multiple return code values. You can then use this success code set for your current task, and reuse it for additional tasks that you create later. This particular example creates a success code set for the zero return code, as illustrated above. You can just use the default return code check box on the first panel to achieve the same goal.

Group tasks

Use the grouping function to run or schedule together a large number of tasks. Instead of creating a command task, define a grouping task. As soon as this type of task is defined, the Group tab is enabled and the following window opens.

On this tab, you can add all of the tasks that you want to run as part of this group. In this example, two tasks have been added to the group. After you have defined all of the tasks, schedule the group and set any success or failure conditions as you would with an individual task.

Schedule tasks

Once you have defined your script and run characteristics of your task, define the schedule on when it will run. You do not have to create a schedule for your task, but to run it you must schedule it either individually or as part of a group.

The schedule tab contains the date and time at which the command is supposed to be run.

You can specify that the command runs once, or at multiple times based on a schedule or on a list of saved schedules. If you want to run multiple tasks on the same date or at the same time, it may be simpler to create one schedule record and use that for each of your tasks rather than recreating it every time.

Set notification

The notification tab is used to tell DB2 where to send the completion code when the task finishes. This does not send the actual results of the task, but only the return code.

Multiple notifications can be sent for an individual task. These notifications can be based on:

- **Task success:** When the task successfully completes, a notification can be sent.
- **Task failure:** When the task does not complete successfully, a notification can be sent.
- **Any condition:** No matter what the result is, a notification is sent.

In addition to the notification level, you can also specify the way in which notification is sent. The task center can either notify contacts or place information in the Journal (see [Journal](#)). The contact list contains a set of users to whom you can send a page or email notifications.

Note that if you want to send messages to users, you must set up an SMTP server on your machine. Without this capability, a message cannot be sent.

The message itself can be customized to include any text that you want. A number of symbolic variables are available within the text that are used to return information about the task:

- **&Categories:** The categories associated with the task.
- **&Completionstatus:** The completion status of the task. This value depends on the success code set associated with the task.
- **&Description:** The description of the task.
- **&Duration:** The length of time the run system took to complete the task from start to finish.
- **&End:** The date and time the task completed.
- **&Howinvoked:** The method used to invoke the task.
- **&Name:** The name of the task.
- **&Owner:** The name of the owner of the task.
- **&Returncode:** The final return code of the task.
- **&Runpartitions:** The partitions on which the task ran.
- **&Runsystem:** The name of the system on which the task ran.
- **&Schedulersystem:** The name of the system on which the task is scheduled.

- **&Start:** The date and time the task began running.
- **&Type:** The task type (that is, whether it is a DB2 script, an OS script, a MVS shell script, a JCL script, or a grouping).
- **&Userid:** The user ID for the task.

Set task actions

The task actions tab determines what happens with the task after it has completed execution.

There are three possible actions that can occur for a task, depending on whether or not it:

- Completes successfully
- Fails at any step
- Results in any outcome (success or failure)

When any of these conditions are met, the task can perform a number of actions. It can:

- Run another task
- Schedule another task
- Disable the schedule of a task
- Delete itself

The task actions section lets you chain together many tasks, each one dependent on the successful completion of the previous task. For example, you can make sure that a table has been successfully created before you define indexes on it.

Set security

The security tab allows you to give read, write, and execute (run) privileges to other users for the task being created. This can be useful when you have a number of different users creating and maintaining tasks. In such a case, it might be easier to create a group of users that are allowed to manage these tasks, rather than giving each user access to the task.

Task list

Once you have created the task, it is displayed in the main task menu.

The items in the task menu are shown by overview category. This is the same category field that you filled in when you were creating the task. By using these categories, you make it a lot simpler to find and manage these tasks.

A task will be removed from this list if it is physically deleted by the user, or if one of the assigned actions was to delete itself after it had finished executing. If the screen is automatically refreshed, the task will eventually disappear once it has run. To determine the status of a task after it has completed, use the Journal, which we'll discuss in the next panel.

Journal

The Journal keeps track of a number of events within the database engine. These events include:

- **Task history:** Displays the results of the tasks that have run
- **Database history:** Displays any maintenance activity against a database
- **Messages:** Lists the error messages that have been produced by the database
- **Notification log:** Contains messages that have been produced by the Health Center or alerts within the system

The Journal can be started from either the Control Center or the tools menu.

The journal can also be started from within the DB2 program group. The initial screen is shown below.

All of the items in the list are tasks (or groups of tasks) that have executed. To get more details on a particular task, double-click on task. For example, clicking the last task in the list illustrated in the figure above shows the report for that task.

The results page shows you whether or not the task was successful and also contains any messages that were produced by the task. There are an additional three tabs that provide more information on the task that was executed. The second tab provides information on the command script that was executed:

The third tab in the task report displays the physical output produced by the task. The output from the SQL that was run is shown in the figure below. Note that the SQL must issue the `connect` statement to the proper database for this to run successfully. Without the connection, the task will not know what database to operate on.

Finally, the last tab indicates the subsequent actions that were taken when the task completed. In this case, the task deleted itself when it ran successfully.

Summary

The Task Center lets you define tasks that can run a variety of commands, including:

- DB2 commands
- Operating system commands
- MVS shell or JCL commands
- Groups of the above types of commands

Tasks can be scheduled to work at specific times, and can notify other users of their own success or failure. In addition, a task can also cause other tasks to be scheduled, unscheduled, or deleted.

Create and manage indexes

Introduction

Indexes are a critical component of any database that you might end up creating. Although the relational model does not require indexes to run queries or calculate results, your end users will be much happier with you if you create some indexes on frequently used tables!

Indexes are physical objects that are associated with individual tables. Any permanent table or declared temporary table can have multiple indexes defined on it. You cannot define an index on a view.

Indexes are used for two primary reasons:

- To ensure uniqueness of data values
- To improve SQL query performance

Indexes can be used to access data in a sorted order more quickly and avoid the time-consuming task of sorting the data using temporary storage. Indexes can also be created on computed columns so that the optimizer can save computation time by using the index instead of doing the calculations. Indexes are maintained automatically by DB2 as data is inserted, updated, and deleted.

How indexes are created

As a DBA, you must be aware of how indexes are created. Although indexes are an optional (but critical!) component of any database, they are sometimes created automatically on your behalf. If you do not plan for indexes in advance, you may not have sufficient resources to create or maintain them.

Indexes can be created through the use of the `CREATE INDEX` command. However, indexes are also automatically generated if you create a table that contains a column with a `UNIQUE` attribute, if a referential constraint is placed against a table, or if a table has been defined with dimension (multidimensional) attributes. All of these conditions require an index to enforce uniqueness in the table, and to provide acceptable performance. Imagine if the database had to scan an entire table each time it needed to check if a value was unique!

In addition to SQL commands that generate indexes, a variety of wizards are available within the DB2 Control Center to help decide which indexes would be most appropriate in a given situation.

Issues with indexes

The first question you'll ask when generating an index is, "Which one will give me the best performance?" Should you create an index on every column within a table to ensure good performance? Or should you only create an index that gives the user direct access to the data?

Indexes take up space on your system. They can reside in either the same table space as the table you are indexing or a separate index table space. Since there are physical limitations in the size of table spaces, you may not have sufficient space to create all of the ones you want.

Updating indexes takes time. Whenever you insert, update, or delete records, DB2 must modify all corresponding indexes that are affected by your actions. Thus, creating 15 indexes on your favorite table will result in 15 index updates every time you change the data. In this case, rather than providing better performance, your indexes will result in *longer* response times.

Finally, indexes may not be appropriate for columns that have few values (low cardinality). An alternate indexing method (multidimensional clustering, or MDC) may be more appropriate.

Where are indexes placed?

Before you decide to create some indexes, you should quickly review the concepts of table spaces.

Tables and indexes are placed into *table spaces*. A table space is used as a layer between the database and the container objects that hold the actual table data.

A *container* is a physical storage device. It can be identified by a directory name, a device name, or a file name. A container is assigned to a table space and a table space can span many containers. The ability to have multiple containers assigned to a table space gets around operating system limitations that may limit the amount of data that one container can have. The relationship among all of these objects is illustrated in the chart below.

Although a table is the basic object that is placed into a table space, you must be aware of additional objects within the DB2 system and how they are mapped to a table space.

Table and index storage

So why the interest in the `CREATE TABLE` command when what you really want to do is create some indexes? The difficulty with index placement is that it is dependent on how the table was defined.

If a table is created without any regard to location, it will end up in the default system space, along with any indexes that are created for it. This means that the `USERSPACE1` table space will quickly fill up with your tables and indexes, since they are both stored in the same place.

Even if you conscientiously placed your tables in separate table spaces, you may still have a problem, since the indexes will be created in the same place. You need to specify the location of your indexes!

Creating tables

When you create a table, you can use an option to specify the table space -- or table spaces -- in which the table and index will be placed:

```
CREATE TABLE TEST (  
    column 1 definition, column 2 definition, ...  
) IN <tablespace name> INDEX IN <index space name>
```

This command gives you the option of specifying where the table is created, along with the index. If you do not specify a separate index table space, the indexes are created in the same table space as the table. You do not have the option of creating indexes in a different table space after the table has been created. The moral of this story: plan ahead before creating your indexes!

If you intend to partition your tables (range partitioning), you have the added flexibility of placing every index for that table into its own table space. This option is only available with partitioned tables.

Creating indexes

Now that you know where your indexes are going to be created, it's time to examine the `CREATE INDEX` command. Here's the basic command syntax:

```
CREATE <UNIQUE> INDEX <index name> ON <table name>
(
  column 1 <ASC | DESC> ,
  column 2 <ASC | DESC> ...
)
```

The `UNIQUE` attribute tells DB2 that the index must enforce uniqueness for all values that are inserted. If a duplicate value is found during an `UPDATE` or `INSERT` command, an error will be returned to the application.

The columns that are listed in brackets are used to generate the index. The optional `ASC` (ascending) and `DESC` (descending) keywords tell DB2 how to order these values in the index itself. These options are useful when you issue SQL statements that sort the results, like so:

```
SELECT * FROM EMPLOYEE
ORDER BY EMPNO DESC
```

If the index has already been created in descending order, DB2 can use the index to return the values in a sorted sequence, rather than having to do a separate sort step. This can save considerable time on large answer sets. If answer sets are sorted in both ascending and descending order, it can be more advantageous to add the additional `ALLOW REVERSE SCANS` to the end of the `CREATE INDEX` command. This tells DB2 to include additional pointers within the index to allow efficient forward and backward chaining within the records. The `ALLOW REVERSE SCANS` is now the default for any indexes created in DB2 9, but prior releases will need to include this option to allow ascending and descending access on the same index.

Including additional columns in an index

DB2 has the ability to add additional columns to the index that you are creating. The `CREATE INDEX` command allows the user to specify columns that are not part of the actual index, but are kept in the index record for performance reasons.

```
CREATE UNIQUE INDEX IX ON EMPLOYEE (EMPNO) INCLUDE (LASTNAME, FIRSTNAME)
```

The index must be `UNIQUE` for columns to be included in the index. When the index is created, the additional columns are added to the index values. The index does not use these values for sorting or determining uniqueness, but can use them when satisfying an SQL query. For example, the following `SELECT` statement would not need to read the actual data rows:

```
SELECT LASTNAME, FIRSTNAME FROM EMPLOYEE WHERE EMPNO < '000300'
```

Examining the Visual Explain for this statement confirms the use of the index to get the answer set.

There are many circumstances in which placing data in the index will help performance. Care should be taken to not use too many columns, however; if you do, the size of the index can begin to approach the size of the physical data itself.

Clustering indexes

A clustering index is an index that is used by DB2 to try and insert records on the same page as other records with similar index key values. If there is no space on that page, an attempt is made to put the record into the surrounding pages.

The advantage of having a clustering index is that `SELECT` statements that look for a particular value (or range of values) on a key can quickly find the answer set without scanning the entire table. Similar key values will be placed on the same data pages so that only a portion of the entire table needs to be read. In addition, the need for table reorganization can also be reduced by using a clustering index.

To insure that sufficient space is available for new rows on existing pages, the `PCTFREE` keyword should be used during table creation to leave some space available for future inserts and updates. Once the table has been loaded, the `PCTFREE` value can be reduced to allow for more records to be added to existing pages.

To create a clustering index on a table, append the `CLUSTER` keyword to the end of the `CREATE INDEX` command, like so:

```
CREATE INDEX DEPTS_IX ON EMPLOYEE(WORKDEPT) CLUSTER
```

The `PCTFREE` keyword can also be used in the creation of an index. Specify a higher `PCTFREE` at index creation time and then set it to a lower value later to allow for records to be inserted into the index without causing index page splits. This is particularly useful in high-transaction environments where there is a lot of insert and delete activity. However, if your database is primarily meant for query workloads, it may be more advantageous to place as much data onto the index pages as possible. In this case, set `PCTFREE` to zero. From a database perspective, fewer page reads will have to be done to load the index, but it will be more expensive to do index maintenance, since index page splits will occur.

How many indexes should you create?

So how many indexes should you create for a table? The answer really depends on the type of application that you are running against the table.

Use the following general rules to determine the typical number of indexes that you define for a table. The number of indexes is based on the primary use of your database:

- For online transaction processing (OLTP) environments, create one or two indexes
- For mixed query and OLTP environments, create between two and five indexes.
- For read-only query environments, create more than five indexes

Another option in determining which indexes to create is to use the Design Advisor within the Control Center. The Design Advisor will ask you a number of questions about your workload and database design and will then determine what the best indexes are.

Referential integrity and indexes

Indexes are a key component of referential integrity. Without indexes, performance would be poor, and integrity checking would be extremely expensive.

Referential integrity allows you to define required relationships between and within tables. The database manager maintains these relationships, which are expressed as referential constraints and require that all values of a given attribute or table column also exist in some other table column. The following figure illustrates an example:

Let's look at some definitions of keys and constraints, using the figure above for illustrative purposes.

A *unique key* is a set of columns in which no two values are duplicated in any other row. Only one unique key can be defined as a primary key for each table. The unique key may also be known as the *parent key* when it is referenced by a foreign key.

A *primary key* is a special case of a unique key. Each table can only have one primary key. In the figure, DEPTNO and EMPNO are the primary keys of the DEPARTMENT and EMPLOYEE tables.

A *foreign key* is a column or set of columns in a table that refer to a unique key or primary key of the same or another table. It is used to establish a relationship with a unique key or primary key and enforces referential integrity among tables. The column WORKDEPT in the EMPLOYEE table is a foreign key because it refers to the primary key -- the column DEPTNO -- in the DEPARTMENT table.

A *parent key* is a primary key or unique key of a referential constraint. A parent table is a table containing a parent key that is related to at least one foreign key in the same or another table. A table can be a parent in an arbitrary number of relationships. In the figure above, the DEPARTMENT table, with a primary key of DEPTNO, is a parent of the EMPLOYEE table, which contains the foreign key WORKDEPT.

A *dependent table* is a table containing one or more foreign keys. A dependent table can also be a parent table. A table can be a dependent in an arbitrary number of relationships. For example, the EMPLOYEE table contains the foreign key WORKDEPT, which is dependent on the DEPARTMENT table that has a primary key.

A *referential constraint* is an assertion that non-null values of a designated foreign key are valid only if they also appear as values of a unique key of a designated parent table. The purpose of referential constraints is to guarantee that database relationships are maintained and data entry rules are followed.

Enforcement of referential constraints has special implications for some SQL operations that depend on whether the table is a parent or a dependent. The database manager enforces referential constraints across systems based on the referential integrity rules. The rules are:

- INSERT
- DELETE
- UPDATE

Creating referential integrity constraints

To define a foreign key relationship between two tables:

1. Create the parent table with a primary key
2. Load data into the parent table
3. Create the dependent table with the foreign key relationship
4. Load data into the dependent table

For the EMPLOYEE and DEPARTMENT example from the previous panel, the definition for the DEPARTMENT table is as follows:

```
CREATE TABLE DEPARTMENT
(
  DEPTNO      INT           NOT NULL PRIMARY KEY,
  DEPTNAME    VARCHAR(20)  NOT NULL,
  MGRNO      INT
)
```

The table definition for the DEPARTMENT table does not necessarily have to have a PRIMARY KEY specification on the DEPTNO line; you can use a CREATE UNIQUE INDEX command instead.

Once data has been loaded into the DEPARTMENT table, the EMPLOYEE table can be created. Note that you don't have to load data into the DEPARTMENT table immediately, but you will need to populate it with data before the EMPLOYEE table can have data inserted into it. The EMPLOYEE table can be defined as follows:

```
CREATE TABLE EMPLOYEE
(
  EMPNO      INT           NOT NULL PRIMARY KEY,
  FIRSTNAME  VARCHAR(20)  NOT NULL,
  LASTNAME   VARCHAR(20)  NOT NULL,
  WORKDEPT   INT           NOT NULL,
  PHONENO    CHAR(12)     NOT NULL,
  FOREIGN KEY(WORKDEPT) REFERENCES DEPARTMENT(DEPTNO)
  ON DELETE RESTRICT)
)
```

The FOREIGN KEY clause defines the relationship between the EMPLOYEE table and the DEPARTMENT table. Because of this relationship, an employee cannot be inserted into the table if his or her WORKDEPT does not already exist in the department table. In addition, any updates to that employee's WORKDEPT will also be checked against the department list. The additional ON DELETE RESTRICT clause tells DB2 to prevent a deletion of a department record in the DEPARTMENT table, unless there are no references to that department in the EMPLOYEE table.

In other words, a department's employees must be all transferred to other departments before DB2 will allow that department to be deleted. Other options are to make the WORKDEPT null upon a department deletion (`ON DELETE SET NULL`), or to delete the employee record that is connected with this record (`ON DELETE CASCADE`).

From an index perspective, both tables should have primary keys defined. This eliminates duplicates from both tables, but also improves select performance. Also, the primary key on the DEPARTMENT table is required so that the foreign key relationship can be established with the EMPLOYEE table.

Referential integrity authority

To create a table with a referential constraint associated with it (foreign key), a user must have appropriate permissions.

Consider our previous example with the EMPLOYEE and DEPARTMENT tables (see [Referential integrity and indexes](#)). If the DEPARTMENT table were owned and managed by the facilities department, and the EMPLOYEE table were managed by the personnel department, the two departments need to cooperate with one another. If the EMPLOYEE table had the foreign constraint defined against the DEPARTMENT table, the users in facilities could not delete any department records without the help of the people in personnel! This is due to the `ON DELETE RESTRICT` rule that was placed against the DEPARTMENT table. This means that a user could place severe restrictions against your table without your knowledge. To prevent this, explicit authority to set referential constraints must be given to the personnel department (the owners of the EMPLOYEE table):

```
GRANT REFERENCES(DEPTNO) ON DEPARTMENT TO USER PERSONNEL
```

Summary

Indexes are a key component of database design. They can be used to improve performance, cluster records on pages, and guarantee uniqueness of records.

Index design is very dependent on the type of workload and the amount of space needed to set aside for storage. Additional fields can be added to an index to improve query performance. Tools also exist to help define the indexes required for your system.

Referential integrity is a form of constraint checking between a dependent table and a parent table. Indexes are required to ensure uniqueness in the parent table, as well as improve performance of key checking during insert and update processing. Special permission must be given to owners of dependent tables to be able to create referential constraints against tables they do not own.

Define data constraints on tables

What are constraints?

The integrity or validity of data in a database is of crucial importance. It is difficult to ensure the validity of data being inserted into a database; DB2 provides the ability to define some rule-based

constraints or checks that can be incorporated into the database. In DB2, the following checks can be used to minimize the risk of inserting incorrect data into a table:

- The fields in a row can be checked to see if they conform to the data type and length of the columns with which they are associated. For example, the value "Geoff" does not match a column data type of INTEGER, and therefore a row with that value will be rejected, thus ensuring the validity of the data in the database.
- If a primary key constraint has been defined on a table, then each row in the table must have a unique value in the column or columns that collectively form the primary key. If a row is inserted with the same key as an existing one, the new row will be rejected.
- If a unique constraint has been defined on a table, each row in the table must comply with this constraint by having a unique value or combination of values that make up the unique key.
- If a foreign key constraint has been defined, each row in the table must have a value in the foreign key column or columns that matches a primary key of a row in a parent table. In some cases, a null value may be acceptable if the column or columns defined as part of the foreign key are also defined as nullable.
- If a check constraint has been defined on a column, each row must comply with the constraint. For example, a check constraint on a salary column of an employee table may prevent an application or user from inserting a new employee record or row for which the salary is less than zero. Any row inserted into the table that has a salary value of less than zero will be rejected, thus minimizing the risk of inserting incorrect data into the table.

Primary key constraints, unique constraints, and foreign key constraints are all discussed in the section on [Create and manage indexes](#). That section deals primarily with check constraints.

Table constraints

Table-check constraints will enforce data integrity at the table level. Once a table-check constraint has been defined for a table, every `UPDATE` and `INSERT` statement will involve checking the restriction or constraint. If the constraint is violated, the row will not be inserted or updated, and an SQL error will be returned.

A table-check constraint can be defined at table creation time or later using the `ALTER TABLE` statement. The table-check constraints can help implement specific rules for the data values contained in the table by specifying the values allowed in one or more columns in every row of a table. This can save time for the application developer, since the validation of each data value can be performed by the database and not by each of the applications accessing the database.

Adding constraints

When you add a check constraint to a table that contains data, there are two possibilities:

- All the rows will meet the check constraint.
- Some or all of the rows do will not meet the check constraint.

In the first case, when all the rows meet the check constraint, the check constraint will be created successfully. Future attempts to insert or update data that does not meet the constraint business rule will be rejected.

If there are some rows that do not meet the check constraint, the check constraint will not be created (i.e., the `ALTER TABLE` statement will fail).

An `ALTER TABLE` statement that adds a new constraint to the `EMPLOYEE` table discussed earlier (see [Referential integrity and indexes](#)) is shown below. The check constraint is named `check_job`. DB2 will use this name to inform us if the constraint is violated when an `INSERT` or `UPDATE` statement fails.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT check_job
  CHECK (JOB IN ('Engineer', 'Sales', 'Manager'))
```

There is no special command used to change a check constraint. Whenever a check constraint needs to be changed, you must drop it and create a new one. Check constraints can be dropped at any time, and this action will not affect your table or the data within it.

Creating tables with constraints

You can add a constraint to a table as you are creating that table. A constraint can be added to individual columns by adding the `CONSTRAINT/CHECK` clause after the column definition:

```
CREATE TABLE EMPLOYEE
(
  EMPNO INT NOT NULL PRIMARY KEY,
  JOB VARCHAR(10) CONSTRAINT CHECK_JOB
    CHECK (JOB IN ('Engineer', 'Sales', 'Manager')),
  ...
)
```

The `CONSTRAINT` name is not required as part of the definition, but it is recommended that you name the constraint in the event that you want to modify it at some later date. If you don't name the constraint, you will have to determine its system-defined name.

Constraints can also be defined across multiple columns, and these definitions are usually placed at the end of the all of the column definitions. These constraints combine column values and are often referred to as *table constraints*. The following SQL is an example of a table constraint that checks an individual's age and salary:

```
CREATE TABLE EMPLOYEE
(
  ...,
  CONSTRAINT CHECK_AGE_SALARY
    CHECK (NOT(AGE < 30 AND SALARY > 60000))
)
```

The unusual logic in the `CHECK` statement is necessary due to the way constraints are handled. The constraint within the brackets must hold true for the record to be inserted. This means that the statement `NOT(AGE < 30 AND SALARY > 60000)` must be true. The translation of this logic is that no one can be less than 30 years old and make more than 60000 a year.

Informational constraints

All of the constraints that we've defined so far are enforced by DB2 when records are inserted or updated. This can lead to high amounts of system overhead, especially when loading large quantities of records.

If an application has already verified information before inserting a record into DB2, it may be more efficient to use *informational constraints*, rather than normal constraints. Informational constraints tell DB2 what format the data should be in, but are not enforced during insert or update processing. However, this information can be used by the DB2 optimizer and may result in better performance of SQL queries. Consider the following `CREATE TABLE` statement:

```
CREATE TABLE EMPDATA
(
  EMPNO INT NOT NULL,
  SEX CHAR(1) NOT NULL
    CONSTRAINT SEXOK
    CHECK (SEX IN ('M', 'F'))
    NOT ENFORCED
    ENABLE QUERY OPTIMIZATION,
  SALARY INT NOT NULL,
    CONSTRAINT SALARYOK
    CHECK (SALARY BETWEEN 0 AND 100000)
    NOT ENFORCED
    ENABLE QUERY OPTIMIZATION
)
```

This example contains two statements that change the behavior of the column constraints. The first option is `NOT ENFORCED`, which instructs DB2 not to enforce the checking of this column when data is inserted or updated. The second option is `ENABLE QUERY OPTIMIZATION`, which is used by DB2 when `SELECT` statements are run against this table. When this value is specified, DB2 will use the information in the constraint when optimizing the SQL.

NOT ENFORCED option

If the table contains the `NOT ENFORCED` option, the behavior of `INSERT` statements may appear odd. The following SQL will not result in any errors when run against the EMPDATA table:

```
INSERT INTO EMPDATA VALUES
(1, 'M', 54200),
(2, 'F', 28000),
(3, 'M', 21240),
(4, 'F', 89222),
(5, 'Q', 34444),
(6, 'K', 132333)
```

Employee number five has a questionable gender (Q), and employee number six has both an unusual gender and a salary that exceeds the limits of the salary column. In both cases DB2 will allow the insert to occur, since the constraints are `NOT ENFORCED`. This points out one of the weaknesses of informational constraints. You must be certain that the data that you are inserting or loading conforms to the definitions that you have placed into DB2.

ENABLE QUERY OPTIMIZATION option

What will probably cause more confusion is the result of a select statement against the EMPDATA table after the insert you ran in the last panel:

```
SELECT * FROM EMPDATA
WHERE SEX = 'Q';

EMPNO      SEX SALARY
-----
0 record(s) selected.
```

DB2 returned the incorrect answer to the query. The value "Q" is found within the table, but the constraint on this column tells DB2 that the only valid values are "M" and "F". The `ENABLE QUERY OPTIMIZATION` keyword also allowed DB2 to use this constraint information when optimizing SQL statements. If this is not the behavior that you want, then you need to change the constraint through the use of the `ALTER` command:

```
ALTER TABLE EMPDATA
ALTER CHECK SEXOK DISABLE QUERY OPTIMIZATION
```

Now, let's re-execute our earlier query. The results are as follows:

```
SELECT * FROM EMPDATA
WHERE SEX = 'Q';

EMPNO      SEX SALARY
-----
5 Q        34444

1 record(s) selected.
```

When should informational constraints be used in DB2? The best scenario for using informational constraints occurs when the user can guarantee that the application program is the only application inserting and updating data. If the application already checks all of the information beforehand, then using informational constraints can result in faster performance and no duplication of effort.

Summary

There are a variety of constraints that are available within DB2 to maintain data integrity. These constraints are data type, primary key, unique, foreign key, and check constraints.

Check constraints allow the user to place rules on data within a column to ensure that certain criteria are met before allowing a row to be inserted. These constraints can be modified to enforce the conditions, or to ignore them. Similarly, the optimizer can also be told to use the information within the constraints for optimization purposes, or to ignore them.

Proper use of check constraints can help to improve query performance and minimize load time. However, without proper data cleansing, the results that are retrieved may not always be accurate.

Create and manage views

Views

Views are virtual tables that are derived from one or more tables or views; they can be used interchangeably with tables when retrieving data. Views can be very useful when you want to hide certain columns or rows of a base table. If you don't want to create another copy of a table, you can use a view to create a virtual table that only shows users the data you want them to see.

When changes are made to data through a view, the data is changed in the underlying table itself. Views themselves do not contain any real data. There are some circumstances under which views cannot be updated, so views can be classified as deletable, updatable, insertable, or read-only. The classification indicates the kind of SQL operations allowed against the view.

A simple view

A simple example will illustrate the power and usefulness of a view. Consider the following personnel table:

```
CREATE TABLE PERSONNEL
(
  PERSON_ID  INT NOT NULL,
  FIRST_NAME VARCHAR(20),
  LAST_NAME  VARCHAR(20),
  SALARY     DEC(9,2),
  EXTENSION  CHAR(4),
  ...
)
```

It's not the most sophisticated of personnel tables, but it will help illustrate a point about views. This table contains information that is highly sensitive, such as employees' salaries. However, much of the information in this table could be used by other departments or users. For example, the extension (phone number) column could be used to produce an internal telephone directory. How can you take advantage of this information without compromising the integrity of the salary information?

You probably guessed that the solution has something to do with views. You can create a view on this table that restricts the user to only seeing certain columns. The following SQL statement creates a view that displays the users' first name, last name, and telephone number:

```
CREATE VIEW TELEPHONE_BOOK AS
(
  SELECT FIRST_NAME, LAST_NAME, EXTENSION FROM PERSONNEL
)
```

Grant users access to this view rather than to the base personnel table. A user issuing a select statement against the view sees only three columns:

```
SELECT * FROM TELEPHONE_BOOK;
```

FIRST_NAME	LAST_NAME	EXTENSION
ANDREW	BAKLARZ	2431
GEOFFREY	BAKLARZ	8734
...		

A view can be much more sophisticated than this, but this example illustrates the fundamental features.

View syntax

Access to the DB2 SQL reference guide would be very useful at this point in time. Of course, you probably don't have that handy, so here is a short syntax diagram for view creation:

```
CREATE VIEW view-name (column list) AS (fullselect)
```

That's not really all of the parts of the command syntax, but it does illustrate what you probably use most often. The `CREATE` statement contains these parts:

- **View-name:** The identifier for this view. It has the same limitations as a real table name and cannot be the same as an existing table.
- **Column list:** This optional portion tells DB2 what the names of the columns should be when the answer set is returned. For example, in the previous example, all of the columns can be renamed, like so:

```
CREATE VIEW TELEPHONE_BOOK(FIRST, LAST, PHONE) AS
(
  SELECT FIRST_NAME, LAST_NAME, EXTENSION FROM PERSONNEL
)
```

- **Fullselect:** This is the SQL that will be used to generate the view definition. A fullselect could return individual rows based on a `WHERE` clause, or it could do joins, aggregations, or any complex SQL operation.

Views with UNION

Views with tables connected through the use of `UNION ALL` have been supported for a number of releases of DB2. `SELECT`, `DELETE`, and `UPDATE` operators have also been allowed, assuming DB2 can determine the table to which the corresponding command is to be applied.

In DB2, support for the `INSERT` operator has been extended to views with `UNION ALL`, as long as the following conditions hold:

- The expressions have the same datatypes
- A constraint exists on at least one column that can be used to uniquely identify where a row should be inserted, and the constraint ranges are non-overlapping

Views defined in this fashion will also support `UPDATE` operations as long as the column being changed does not violate the constraint for that column. In this case, the user must first `DELETE` and then `INSERT` the record.

Deletable views

Depending on how a view is defined, the view can be *deletable*. A deletable view is a view against which you can successfully issue a `DELETE` statement. There are a few rules that need to be followed for a view to be considered deletable:

- Each `FROM` clause of the outer fullselect must identify only one base table (with no `OUTER` clause), deletable view (with no `OUTER` clause), deletable nested table expression, or deletable common table expression
- The outer fullselect must not use the `VALUES` clause
- The outer fullselect must not use the `GROUP BY` or `HAVING` clauses
- The outer fullselect must not include column functions in its select list
- The outer fullselect must not use set operations (`UNION`, `EXCEPT`, or `INTERSECT`) with the exception of `UNION ALL`
- The base tables in the operands of a `UNION ALL` must not be the same table, and each operand must be deletable
- The select list of the outer fullselect must not include `DISTINCT`

A view must meet all the rules listed above to be considered a deletable view.

Updatable views

An *updatable* view is a special case of a deletable view. A deletable view becomes an updatable view when at least one of its columns is updatable. A column of a view is updatable when all of the following rules are true:

- The view must be deletable
- The column must resolve to a column of a table (not using a dereference operation), and the `READ ONLY` option must not be specified
- All the corresponding columns of the operands of a `UNION ALL` must have exactly matching data types (including length or precision and scale), and matching default values if the fullselect of the view includes a `UNION ALL`

Insertable and read-only views

Insertable views allow you to insert rows using the view definition. A view is insertable when all of its columns are updatable. For example, consider the following `PERSONNEL` table and its associated view, `TELEPHONE_BOOK`:

```
CREATE TABLE PERSONNEL
(
  PERSON_ID   INT NOT NULL,
  FIRST_NAME  VARCHAR(20) NOT NULL,
  LAST_NAME   VARCHAR(20) NOT NULL,
  SALARY      DEC(9,2) NOT NULL,
  EXTENSION   CHAR(4) NOT NULL
)

CREATE VIEW TELEPHONE_BOOK AS
(
  SELECT PERSON_ID, FIRST_NAME, LAST_NAME, EXTENSION FROM PERSONNEL
)
```

The TELEPHONE_BOOK view is not insertable because an insert statement does not include the SALARY field, and this field cannot be null. However, if the original table definition included a DEFAULT clause for the SALARY field, or if the field was allowed to be null, then the view would be insertable.

A *read-only* view is a nondeletable view (see [Deletable views](#)). A view can be read-only if it does *not* comply with *at least one* of the rules for deletable views.

In the event that a view is read-only, an `INSTEAD OF` trigger can be defined against it to direct how an insert should take place.

An `INSTEAD OF` trigger is used only on *views*, not base tables. It has similar characteristics to a normal trigger, but has the following restrictions:

- It is only allowed on views
- It is always `FOR EACH ROW`
- `DEFAULT` values get passed as null
- It cannot use positioned `UPDATE/DELETE` on cursor over view with `INSTEAD OF UPDATE/DELETE` trigger

Define an `INSTEAD OF` trigger to handle situations in which an insert is ambiguous, and then get around the limitations of a read-only view.

WITH CHECK OPTION

If the view definition includes conditions (such as a `WHERE` clause) and its intent is to ensure that any `INSERT` or `UPDATE` statements referencing the view will have the `WHERE` clause applied, the view must be defined using `WITH CHECK OPTION`. This option can ensure the integrity of the data being modified in the database. An SQL error will be returned if the condition is violated during an `INSERT` or `UPDATE` operation.

The following is an example of a view definition using `WITH CHECK OPTION`. `WITH CHECK OPTION` is required to ensure that the condition is always checked. In this case, you want to ensure that the DEPT is always 10. This will restrict the input values for the DEPT column. When a view is used to insert a new value, the `WITH CHECK OPTION` is always enforced.

```
CREATE VIEW EMP_VIEW2
  (EMPNO, EMPNAME, DEPTNO, JOBTITLE, HIREDATE)
AS SELECT ID, NAME, DEPT, JOB, HIREDATE FROM EMPLOYEE
   WHERE DEPT=10
   WITH CHECK OPTION
```

If this clause did not exist, it would be possible for someone working with this view to update a record so that it is no longer part of the view. For example, the following SQL statement would cause some problems.

```
UPDATE EMP_VIEW2 SET DEPT=20 WHERE DEPT=10
```

The results of this statement is that the view now contains no records, since there are no more employees in department 10.

Inoperative views

An *inoperative* view is a view that is no longer available for SQL statements. A view becomes inoperative if:

- A privilege on which the view definition is dependent is revoked.
- An object, such as a table, alias, or function, on which the view definition is dependent is dropped.
- A view on which the view definition is dependent becomes inoperative.
- A view that is the superview of the view definition (the subview) becomes inoperative.

A view cannot be altered in DB2. You must recreate it with the changes that you want.

Summary

A view is an efficient way of representing data without needing to maintain it. A view is not an actual table and requires no permanent storage.

A view can include all or some of the columns or rows contained in the tables on which it is based. For example, you can join a department table and an employee table in a view, so that you can list all employees in a particular department.

A view can include an option to guarantee that inserts and updates on the view do not violate the view definition.

Views can allow for inserts, deletes, and updates against the base table as long as certain criteria are met. Even if a view cannot be updated, an `INSTEAD OF` trigger may be written to get around the restriction.

Access system catalog tables

System catalog tables

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

The system catalog tables are like any other table found in the database. You can `SELECT` information from them using standard SQL syntax. A sample listing of catalog tables is found in the following illustration:

These tables are updated during the operation of a database -- when a table is created, for example. You cannot explicitly create or drop these tables, but you can query and view their

content. When the database is created, in addition to the system catalog table objects, a number of other database objects are defined in the system catalog:

- A set of routines (functions and procedures) is created in the schemas SYSIBM, SYSFUN, and SYSPROC.
- A set of read-only views for the system catalog tables is created in the SYSCAT schema.
- A set of updatable catalog views is created in the SYSSTAT schema. These updatable views allow you to update certain statistical information to investigate the performance of a hypothetical database, or to update statistics without using the RUNSTATS utility.

After a database has been created, you may wish to limit the access to the system catalog views.

Privileges on the system catalog tables

During database creation, SELECT privilege on the system catalog views is granted to PUBLIC. In most cases, this does not present any security problems. However, these tables describe every object in the database, and you may not want everyone to know these details.

To reduce any security risks, you should revoke SELECT privilege from PUBLIC and then grant the SELECT privilege as required to specific users. Granting and revoking the SELECT privilege on the system catalog views is done in the same way as for it is for any other view, but you must have either SYSADM or DBADM authority to do it.

The following is a set of tables to which you should restrict access:

- SYSCAT.DBAUTH
- SYSCAT.TABAUTH
- SYSCAT.PACKAGEAUTH
- SYSCAT.INDEXAUTH
- SYSCAT.COLAUTH
- SYSCAT.PASSTHRUAUTH
- SYSCAT.SCHEMAAUTH

Restricting access to these tables prevents information on user privileges from becoming available to everyone with access to the database.

In DB2 9, another option exists for restricting access to system catalog tables. When issuing the CREATE DATABASE command, the RESTRICTIVE option can be used which results in no privileges automatically granted to PUBLIC.

Revoking SELECT access

To remove SELECT access from one of the system catalog tables, issue the following command from a DB2 command line (you must have SYSADM or DBADM authority to do so):

```
REVOKE SELECT ON SYSCAT.DBAUTH FROM PUBLIC
```

You should revoke PUBLIC SELECT access from all of the system tables that you feel a user should not be able to view.

Useful catalog tables

Of course, every DB2 catalog table is useful, but there are a few of them that you may particularly want to query. There are well over 60 system catalog tables and these are described in detail in Volume 1 of the DB2 SQL Reference manual. Some especially useful views are:

- **SYSCAT.COLUMNS:** Contains one row for each column (including inherited columns, where applicable) that is defined for a table or view.
- **SYSCAT.INDEXCOLUSE:** Lists all columns that participate in an index.
- **SYSCAT.INDEXES:** Contains one row for each index (including inherited indexes, where applicable) that is defined for a table.
- **SYSCAT.TABLES:** Contains one row for each table, view, nickname, or alias that is created. All of the catalog tables and views have entries in the SYSCAT.TABLES catalog view.
- **SYSCAT.VIEWS:** Contains one or more rows for each view that is created.

For example, if you want to determine what the columns are within the EMPLOYEE table, along with their datatype, length, and scale, then you could run the following SQL to retrieve this information:

```
SELECT COLNAME, TYPENAME, LENGTH, SCALE FROM SYSCAT.COLUMNS  
WHERE TABNAME='EMPLOYEE'
```

Of course, you could just use the Control Center to view this information, but a SELECT statement is useful when creating a script that generates this information for a number of tables. The SYSCAT.VIEWS view is also particularly useful to determine the state of the views in the database.

Summary

The system catalog tables contain information about the definitions of the database objects and security information about the type of access that users have to these objects.

All of the system catalog tables have PUBLIC select access. For higher levels of security, the DBA may wish to revoke PUBLIC access to these objects.

Finally, system catalog tables contain useful information that you can retrieve by using standard SQL select statements. Using SQL statements makes it easy to create scripts that retrieve information about a large number of tables, columns, indexes, and other objects that need to be maintained.

Enforce data uniqueness

Enforcing data uniqueness

Many applications require that the data within a table be unique. This uniqueness usually only applies to certain columns within the row, such as an employee ID or vendor number. This uniqueness guarantees that you have only one record that represents each individual user, transaction, or row.

The database also needs the ability to enforce unique rows within a table. To implement referential integrity, the parent table must have unique rows; otherwise, the relationship between the child and the parent rows would be ambiguous. In addition, without this form of uniqueness and referential integrity, DB2 could not optimize the access to multiple tables in a star-schema format. Clearly, the ability to have unique rows is critical to many database applications.

Creating unique records

You can guarantee that rows within a table are unique in a variety of ways:

Use a primary key. During table creation, you can specify that a column is the primary key of the table:

```
CREATE TABLE EMPLOYEE
(
  EMPNO INT NOT NULL PRIMARY KEY,
  LASTNAME VARCHAR(20) NOT NULL,
  ...
)
```

The `PRIMARY KEY` clause on the column definition tells DB2 to generate an index automatically that will enforce the uniqueness of this column. In addition, there is only one primary key for the entire table, so no other column can contain this clause. If multiple columns are required to guarantee uniqueness of the row, the `PRIMARY KEY` clause must follow the table definition:

```
CREATE TABLE EMPLOYEE
(
  EMPNO INT NOT NULL,
  LASTNAME VARCHAR(20) NOT NULL,
  ...,
  PRIMARY KEY (EMPNO, LASTNAME)
)
```

Use the `UNIQUE` clause. The `UNIQUE` clause can also be used to generate unique values within a row. Every column with this clause will be unique:

```
CREATE TABLE EMPLOYEE
(
  EMPNO INT NOT NULL UNIQUE,
  SOCINS CHAR(11) NOT NULL UNIQUE,
  ...
)
```

In this example, both the employee number and the social insurance number must be unique. In other words, there should be no two people with the same employee number, and no two people with the same social insurance number. To enforce this uniqueness, DB2 generates two indexes automatically, one for each column.

Use a unique index. Finally, uniqueness can also be guaranteed through the use of a unique index. The `CREATE INDEX` command has the option of specifying that the values must be unique in the column (or columns) that are being indexed:

```
CREATE UNIQUE INDEX UNIQUE_EMPLOYEE ON EMPLOYEE(EMPNO)
```

This index should be created immediately after creating the table. Otherwise, there is the possibility that records will be inserted into the table that are not unique. If this is the case, the index creation will fail because of duplicate records.

Eliminating duplicate rows

There may be situations in which some data within a table is not unique. This can be due to "dirty" data, the result of consolidating many data sources, or the columns returned in the result set. Since the data cannot be made unique, you must resort to techniques with SQL that will eliminate these duplicate rows.

The simplest way of eliminating duplicate rows from a result set is to use the `DISTINCT` keyword within a select statement:

```
SELECT DISTINCT WORKDEPT FROM EMPLOYEE
```

For multiple answer sets that are joined together, the user should specify `UNION` rather than `UNION ALL`. `UNION ALL` does not eliminate duplicates in the answer set, while `UNION` does.

```
SELECT WORKDEPT FROM EMPLOYEE WHERE EMPNO > '000100'  
UNION  
SELECT WORKDEPT FROM EMPLOYEE WHERE EMPNO < '000900'
```

Summary

Uniqueness can be guaranteed in tables by using primary keys, `UNIQUE` clauses, or unique indexes. Uniqueness in answer sets can also be guaranteed by using the `DISTINCT` clause in a select statement, or the `UNION` clause with multiple answer sets.

Conclusion

Summary

This tutorial has covered the following topics with respect to database management:

1. Ability to create DB2 tasks using the GUI tools
2. Knowledge of the creation and management of indexes
3. Ability to create constraints on tables (e.g., RI, informational, unique)
4. Ability to create views on tables
5. Skill in examining the contents of the system catalog tables
6. Knowledge of how to enforce data uniqueness

Although the material presented in this tutorial has given you a good overview of indexes, constraints, referential integrity, and views, nothing prepares you more for certification than actually trying out these commands yourself and working on a real database. While many of these features are not needed to run a database, their use will result in better performance and improved control over the quality of your data.

© Copyright IBM Corporation 2006

(www.ibm.com/legal/copytrade.shtml)

Trademarks

(www.ibm.com/developerworks/ibm/trademarks/)