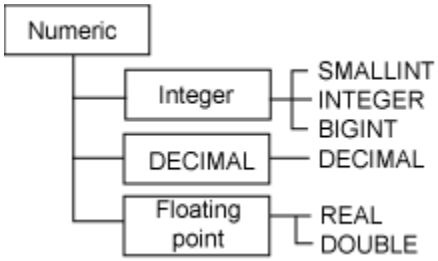
Data types

Databases provide a rich and flexible assortment of data types. They come with basic data types such as `INTEGER`, `CHAR`, and `DATE`. It also includes facilities to create user-defined data types (UDTs) so that you can create complex, nontraditional data types suited to today's complex programming environments. Choosing which type to use in a given situation depends on the type and range of information that is stored in the column.

There are four categories of built-in data types: numeric, string, datetime, and XML.

The user-defined data types are categorized as: distinct, structured, and reference.

Numeric data types

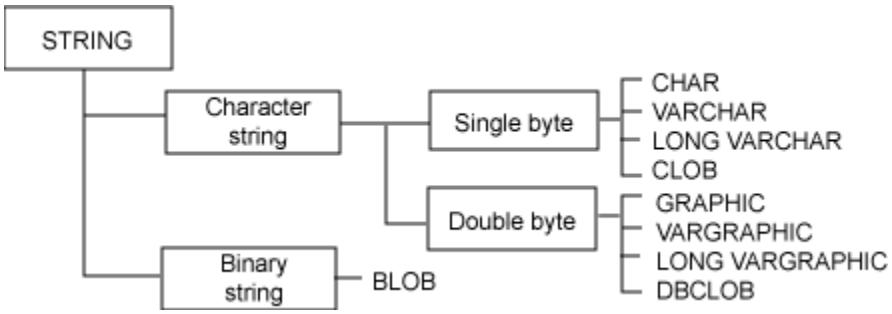


There are three categories of numeric data types, as diagrammed in the preceding figure. These types vary in the range and precision of numeric data they can store.

- *Integer*: `SMALLINT`, `INTEGER`, and `BIGINT` are used to store integer numbers. For example, an inventory count could be defined as `INTEGER`. `SMALLINT` can store integers from -32,768 to 32,767 in 2 bytes. `INTEGER` can store integers from -2,147,483,648 to 2,147,483,647 in 4 bytes. `BIGINT` can store integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 in 8 bytes.
- *Decimal*: `DECIMAL` is used to store numbers with fractional parts. To define this data type, specify a *precision* (p), which indicates the total number of digits, and a *scale* (s), which indicates the number of digits to the right of the decimal place. A column defined by `DECIMAL`(10,2) that held currency values could hold values up to 99999999.99 dollars. The amount of storage required in the database depends on the precision and is calculated by the formula p/2 +1. So, `DECIMAL`(10,2) would require 10/2 + 1 or 6 bytes.
- *Floating point*: `REAL` and `DOUBLE` are used to store approximations of numbers. For example, very small or very large scientific measurements could be defined as `REAL`. `REAL` can be defined with a length between 1 and 24 digits and requires 4 bytes of storage. `DOUBLE` can be defined with a length of between 25 and 53 digits and requires 8 bytes of storage. `FLOAT` can be used as a synonym for `REAL` or `DOUBLE`.

Back to top

String data types



DATABASES provides several data types for storing character data or strings, as diagrammed in the preceding figure. Choose a data type based on the size of the string you are going to store and what data will be in the string.

The following data types are used to store single-byte character strings:

- `CHAR` or `CHARACTER` is used to store fixed-length character strings up to 254 bytes. For example, a manufacturer may assign an identifier to a part with a specific length of eight characters, and therefore store that identifier in the database as a column of `CHAR(8)`.
- `VARCHAR` is used to store variable-length character strings. For example, a manufacturer may deal with a number of parts with identifiers of different lengths, and thus store those identifiers as a column of `VARCHAR(100)`. The maximum length of a `VARCHAR` column is 32,672 bytes. In the database, `VARCHAR` data only takes as much space as required.

The following data types are used to store double-byte character strings:

- `GRAPHIC` is used to store fixed-length double-byte character strings. The maximum length of a GRAPHIC column is 127 characters.
- `VARGRAPHIC` is used to store variable-length double-byte character strings. The maximum length of a VARGRAPHIC column is 16,336 character

Databases also provides data types to store very long strings of data. All long string data types have similar characteristics. First, the data is not stored physically with the row data in the database, which means that additional processing is required to access this data. Long data types can be defined up to 2GB in length. However, only the space required is actually used. The long data types are:

- LONG VARCHAR
- CLOB (character large object)
- LONG VARGRAPHIC
- DBCLOB (double-byte character large object)

- BLOB (binary large object)

Datetime data types

DATABASES provides three data types to store dates and times:

- DATE
- TIME
- TIMESTAMP

The values of these data types are stored in the database in an internal format; however, applications can manipulate them as strings. When one of these data types is retrieved, it is represented as a character string. Enclose the value in quotation marks when updating these data types.

DATABASES provide built-in functions to manipulate datetime values. For example, you can determine the day of the week of a date value using the `DAYOFWEEK` or `DAYNAME` functions. Use the `DAYS` function to calculate how many days lie between two dates. DATABASES also provide special registers to generate the current date, time, or timestamp based on the time-of-day clock. For example, `CURRENT DATE` returns a string representing the current date on the system.

The format of the date and time values depends on the country code of the database, which is specified when the database is created. There are several formats available: ISO, USA, EUR, and JIS. For example, if your database is using the USA format, the format of date values would be `mm/dd/yyyy`. You can change the format by using the `DATETIME` option of the `BIND` command when creating your application.

There is a single format for the TIMESTAMP data type. The string representation is `yyyy-mm-dd-hh.mm.ss.nnnnnn`.

Tables

All data is stored in tables in the database. A *table* consists of one or more columns of various data types. The data is stored in rows or records.

Tables are defined using the `CREATE TABLE` SQL statement. DATABASES also provides a GUI tool, the DATABASES Control Center, for creating tables, which creates a table based on information you specify. It also generates the `CREATE TABLE` SQL statement, which can be used in a script or application program at a later time.

A database has a set of tables, called the *system catalog tables*, which hold information about all the objects in the database. DATABASES provides views for the base system catalog tables. Look at the catalog views using `SELECT` statements, just like any other

table in the database; however, you cannot modify the data using INSERT, UPDATE, or DELETE statements. The tables are automatically updated as a result of data definition statements (DDL), such as CREATE, and other operations, such as RUNSTATS.

Creating a table

Use the CREATE TABLE SQL statement to define a table in the database. The following statement creates a simple table named BOOKS that contains three columns:

```
CREATE TABLE BOOKS ( BOOKID INTEGER,
                     BOOKNAME VARCHAR(100),
                     ISBN CHAR(10) )
```

You can also use the CREATE TABLE SQL statement to create a table that is like another table or view in the database:

```
CREATE TABLE MYBOOKS LIKE BOOKS
```

This statement creates a table with the same columns as the original table or view. The columns of the new table have the same names, data types, and nullability attributes as the columns in the old one. You can also specify clauses that copy other attributes, like column defaults and identify attributes.

There are many options available for the CREATE TABLE statement (they are presented in the following sections as new concepts are introduced). The details of the CREATE TABLE SQL statement can be found in the SQL Reference.

Once you've created a table, there are several ways to populate it with data. The INSERT statement lets you insert a row or several rows of data into the table. DATABASES also provides utilities to insert large amounts of data from a file. The IMPORT utility inserts rows using INSERT statements. It is designed for loading small amounts of data into the database. The LOAD utility, intended for loading large volumes of data, inserts rows directly onto data pages in the database and is much faster than the IMPORT utility.

Storing tables in the database

Tables are stored in the database in *tablespaces.* Tablespaces have physical space allocated to them. Create the tablespace before creating the table.

When you create a table, let DATABASES place the table in a default tablespace or specify the tablespace in which you'd like the table to reside. The following CREATE TABLE statement places the BOOKS table in the BOOKINFO tablespace:

```
CREATE TABLE BOOKS ( BOOKID INTEGER,
                     BOOKNAME VARCHAR(100),
                     ISBN CHAR(10) )
             IN BOOKINFO
```

Although tablespaces are not covered in detail here, it is important to understand that defining tablespaces appropriately has an effect on the performance and maintainability of the database

---

Altering a table

Use the `ALTER TABLE` SQL statement to change characteristics of a table. For instance, you can add or drop:

- A column
- A primary key
- One or more unique or referential constraints
- One or more check constraints

The following statement adds a column called BOOKTYPE to the BOOKS table:

```
ALTER TABLE BOOKS ADD BOOKTYPE CHAR(1)
```

You can also change characteristics of specific columns in a table:

- The identity attributes of a column
- The length of a string column
- The datatype of a column
- The nullability of a column
- The constraint of a column

There are restrictions on altering columns:

- When altering the length of a string column, you can only increase the length.
- When altering the datatype of a column, the new datatype must be compatible with the existing data type. For example, you can convert CHAR columns to VARCHAR columns, but cannot convert them to GRAPHIC or numeric columns. Numeric columns can be converted to any other numeric data type as long as the new datatype is large enough to hold the values. For example, convert an INTEGER column to BIGINT, but, a DECIMAL(10,2) column cannot be converted to SMALLINT.

- Fixed length strings can be converted to variable length string and variable length strings can be converted to fixed length. For example, a CHAR(100) can be converted to VARCHAR(150). Similar restrictions exist for variable length graphic strings.

The following statement changes the DATATYPE of column BOOKNAME from VARCHAR(100) to VARCHAR(200) and changes the nullability of the ISBN column to NOT NULL:

```
ALTER TABLE BOOKS ALTER BOOKNAME SET DATA TYPE VARCHAR(200) ALTER ISBN SET
NOT NULL
```

Certain characteristics of a table cannot be changed. For example, you cannot change the tablespace in which the table resides, the order of columns, or change the datatype of some columns. To change characteristics such as these, save the table data, drop the table, and recreate it.

Dropping a table

The `DROP TABLE` statement removes a table from the database, deleting the data and the table definition. If there are indexes or constraints defined on the table, they are dropped as well.

The following `DROP TABLE` statement deletes the BOOKS table from the database:

```
DROP TABLE BOOKS
```

NOT NULL, DEFAULT, and GENERATED column options

The columns of a table are specified in the `CREATE TABLE` statement by a column name and data type. The columns can have additional clauses specified that restrict the data in the column.

By default, a column allows null values. If you do not want to allow null values, specify the `NOT NULL` clause for the column. Specify a default value using the `WITH DEFAULT` clause and a default value. The following `CREATE TABLE` statement creates a table called BOOKS, where the BOOKID column does not allow null values and the default value for BOOKNAME is TBD:

```
CREATE TABLE BOOKS ( BOOKID INTEGER NOT NULL,
                     BOOKNAME VARCHAR(100) WITH DEFAULT 'TBD',
```

```
                        ISBN CHAR(10) )
```

In the BOOKS table, the BOOKID is a unique number assigned to each book. Rather than have the application generate the identifier, you can specify that DATABASES is to generate a BOOKID using the `GENERATED ALWAYS AS IDENTITY` clause:

```
CREATE TABLE BOOKS ( BOOKID INTEGER NOT NULL AUTOINCREMENT,
                     BOOKNAME VARCHAR(100) WITH DEFAULT 'TBD',
                     ISBN CHAR(10) )
```

`GENERATED ALWAYS AS IDENTITY` causes a BOOKID to be generated for each record. The first value generated is 1 and succeeding values are generated by incrementing the previous value by 1.

Also use the `GENERATED ALWAYS` option to have DATABASES calculate the value of a column automatically. The following example defines a table called AUTHORS, with columns FICTIONBOOKS and NONFICTIONBOOKS that hold counts for fiction and nonfiction books, respectively. The TOTALBOOKS column is calculated by adding the FICTIONBOOKS and NONFICTIONBOOKS columns:

```
CREATE TABLE AUTHORS (AUTHORID INTEGER NOT NULL PRIMARY KEY,
                      LNAME VARCHAR(100),
                      FNAME VARCHAR(100),
                      FICTIONBOOKS INTEGER,
                      NONFICTIONBOOKS INTEGER,
                      TOTALBOOKS INTEGER GENERATED ALWAYS
                                 AS (FICTIONBOOKS +
NONFICTIONBOOKS) )
```

Constraints

DATABASES provides several ways to control what data can be stored in a column. These features are called *constraints* or *rules* that the database manager enforces on a data column or set of columns.

DATABASES provides three types of constraints: unique, referential integrity, and table check.

The following sections provide detailed descriptions of each type of constraint.

Unique constraints

Unique constraints are used to ensure that values in a column are unique. Unique constraints can be defined over one or more columns. Each column included in the unique constraint must be defined as `NOT NULL`.

Unique constraints can be defined either as the `PRIMARY KEY` or `UNIQUE` constraint. These are defined when a table is created as part of the `CREATE TABLE` SQL statement or added after the table is created using the `ALTER TABLE` statement.

When do you define a `PRIMARY KEY`, and when do you define a `UNIQUE` key? This depends on the nature of the data. In the previous example, the BOOKS table has a BOOKID column which is used to uniquely identify a book. This value is also used in other tables that contain information related to this book. In this case, you would define BOOKID as a primary key. DATABASES allows only one primary key to be defined on a table.

The ISBN number column needs to be unique but is not a value that is otherwise referenced in the database. In this case, the ISBN column is defined as `UNIQUE`:

```
CREATE TABLE BOOKS (BOOKID INTEGER NOT NULL PRIMARY KEY,
                    BOOKNAME VARCHAR(100),
                    ISBN CHAR(10) NOT NULL CONSTRAINT BOOKSISBN UNIQUE )
```

The `CONSTRAINT` keyword lets you specify a name for the constraint. In this example, the name of the unique constraint is BOOKSISBN. Use this name in the `ALTER TABLE` statement if you want to drop the specific constraint.

DATABASES lets you define only one primary key on a table; however, you can define multiple unique constraints.

Whenever you define a `PRIMARY KEY` or `UNIQUE` constraint on a column, DATABASES creates a unique index to enforce uniqueness on the column. DATABASES does not let you create more than one unique index defined on the same columns. Therefore, you cannot define a PRIMARY KEY and UNIQUE constraint on the same columns. For example, both of the following statements against the BOOKS table fail because a PRIMARY KEY already exists:

```
ALTER TABLE BOOKS ADD  CONSTRAINT UNIQUE (BOOKID)
CREATE UNIQUE INDEX IBOOKS ON BOOKS (BOOKID)
```

Referential integrity constraints

Referential integrity constraints are used to define relationships between tables and ensure that these relationships remain valid. Suppose you have one table that holds information about authors and another table that lists the books that those authors have written. There is a relationship between the BOOKS table and the AUTHORS table --

each book has an author and that author must exist in the AUTHOR table. Each author has a unique identifier stored in the AUTHORID column. The AUTHORID is used in the BOOKS table to identify the author of each book. To define this relationship, define the AUTHORID column of the AUTHORS table as a primary key and then define a foreign key on the BOOKS table to establish the relationship with the AUTHORID column in the AUTHORS table:

```
CREATE TABLE AUTHORS (AUTHORID INTEGER NOT NULL PRIMARY KEY,
                      LNAME VARCHAR(100),
                      FNAME VARCHAR(100))
CREATE TABLE BOOKS (BOOKID INTEGER NOT NULL PRIMARY KEY,
                    BOOKNAME VARCHAR(100),
                    ISBN CHAR(10),
                    AUTHORID INTEGER REFERENCES AUTHORS)
```

The table that has a primary key that relates to another table -- AUTHOR, here -- is called a *parent table.* The table to which the parent table relates -- BOOKS, here -- is called a *dependent table.* You may define more than one dependent table on a single parent table.

You can also define relationships between rows of the same table. In such a case, the parent table and dependent tables are the same table.

When you define referential constraints on a set of tables, DATABASES enforces referential integrity rules on those tables when update operations are performed against them:

- DATABASES ensure that only valid data is inserted into columns where referential integrity constraints are defined. This means that you must always have a row in the parent table with a key value that is equal to the foreign key value in the row that you are inserting into a dependent table. For example, if a new book is being inserted into the BOOKS table with an AUTHORID of 437, then there must already be a row in the AUTHORS table where AUTHORID is 437.
- DATABASES also enforce rules when rows that have dependent rows in a dependent table are deleted from a parent table. The action DATABASES takes depends on the delete rule defined on the table. There are four rules that can be specified: RESTRICT, NO ACTION, CASCADE and SET NULL.
    - If RESTRICT or NO ACTION is specified, DATABASES does not allow the parent row to be deleted. The rows in dependent tables must be deleted before the row in the parent table. This is the default, so this rule applies to the AUTHORS and BOOKS tables as defined so far.
    - If CASCADE is specified, then deleting a row from the parent table automatically also deletes dependent rows in all dependent tables.
    - If SET NULL is specified, then the parent row is deleted from the parent table and the foreign key value in the dependent rows is set to null (if nullable).
- When updating key values in the parent table, there are two rules that can be specified: RESTRICT and NO ACTION. RESTRICT does not allow a key value

to be updated if there are dependent rows in a dependent table. NO ACTION causes the update operation on a parent key value to be rejected if, at the end of the update, there are dependent rows in a dependent table that do not have a parent key in the parent table.

Table check constraints

Table check constraints are used to verify that column data does not violate rules defined for the column and to restrict the values in a certain column of a table. DATABASES ensures that the constraint is not violated during inserts and updates.

Suppose that you add a column to the BOOKS table for a book type, and the values that you wish to allow are F (fiction) and N (nonfiction). You can add a column BOOKTYPE with a check constraint as follows:

```
ALTER TABLE BOOKS ADD BOOKTYPE CHAR(1) CHECK (BOOKTYPE IN
('F','N') )
```

You can define check constraints when you create the table or add them later using the `ALTER TABLE` SQL statement. You can modify check constraints by dropping and then recreating them using the `ALTER TABLE` SQL statement.

Views

Views allow different users or applications to look at the same data in different ways. This not only makes the data simpler to access, but it can also be used to restrict which rows and columns users view or update.

For example, suppose that a company has a table containing information about its employees. A manager needs to see address, telephone number, and salary information about his employees only, while a directory application needs to see all employees in the company along with their address and telephone numbers, but not their salaries. You can create one view that shows all the information for the employees in a specific department and another that shows only the name, address, and telephone number of all employees.

To the user, a view just looks like a table. Except for the view definition, a view does not take up space in the database; the data presented in a view is derived from another table. You can create a view on an existing table (or tables), on another view, or some combination of the two. A view defined on another view is called a *nested view*.

You can define a view with column names that are different than the corresponding column names of the base table. You can also define views that check to see if data inserted or updated stays within the conditions of the view.

Indexes

An *index* is an ordered list of the key values of a column or columns of a table. There are two reasons why you might create an index:

- To ensure uniqueness of values in a column or columns.
- To improve performance of queries against the table. The DATABASES optimizer uses indexes to improve performance when performing queries or to present results of a query in the order of the index.

Indexes can be defined as unique or nonunique. *Nonunique* indexes allow duplicate key values; *unique* indexes allow only one occurrence of a key value in the list. Unique indexes do allow a single null value to be present. However, a second null value would cause a duplicate and therefore is not allowed.

Indexes are created using the `CREATE INDEX` SQL statement. Indexes are also created implicitly in support of a `PRIMARY KEY` or `UNIQUE` constraint. When a unique index is created, the key data is checked for uniqueness and the operation fails if duplicates are found.

Indexes are created as ascending, descending, or bidirectional. The option you choose depends on how the application accesses the data.

Creating indexes

In the example, you have a primary key on the BOOKID column. Often, users conduct searches on the book title, so an index on BOOKNAME would be appropriate. The following statement creates a nonunique ascending index on the BOOKNAME column:

```
CREATE INDEX IBOOKNAME ON BOOKS (BOOKNAME)
```

The index name, IBOOKNAME, is used to create and drop the index. Other than that, the name is not used in queries or updates to the table.

By default, an index is created in ascending order, but you can also create indexes that are descending. You can even specify different orders for the columns in the index. The following statement defines an index on the AUTHORID and BOOKNAME columns. The values of the AUTHORID column are sorted in descending order, and the values of the BOOKNAME column are sorted in ascending order within the same AUTHORID:

```
CREATE INDEX I2BOOKNAME ON BOOKS (AUTHOID DESC, BOOKNAME ASC)
```

When an index is created in a database, the keys are stored in the specified order. The index helps improve the performance of queries requiring the data in the specified order. An ascending index is also used to determine the result of the `MIN` column function; a

descending index is used to determine the result of the MAX column function. If the application requires the data to be ordered in the opposite sequence to the index as well, DATABASES allows the creation of a bidirectional index. A *bidirectional* index eliminates the need to create an index in the reverse order, and it eliminates the need for the optimizer to sort the data in the reverse order. It also allows the efficient retrieval of MIN and MAX functions values. To create a bidirectional index, specify the ALLOW REVERSE SCANS option on the CREATE INDEX statement:

```
CREATE INDEX BIBOOKNAME ON BOOKS (BOOKNAME) ALLOW REVERSE
SCANS
```

DATABASES does not let you create multiple indexes with the same definition. This applies even to indexes that you create implicitly in support of a primary key or unique constraint. Because the BOOKS table already has a primary key defined on the BOOKID column, attempting to create an index on BOOKID column fails.

Creating an index can take a long time. DATABASES reads each row to extract the keys, sort those keys, and then write the list to the database. If the table is large, then a temporary tablespace is used sort the keys.

The index is stored in a tablespace. If your table resides in a database-managed tablespace, you have the option of separating the indexes into a separate tablespace. Define this when you create the table, using the INDEXES IN clause. The location of a table's indexes is set when the table is created and cannot be changed unless the table is dropped and recreated.

DATABASES also provides the DROP INDEX SQL statement to remove an index from the database. There is no way to modify an index. If you need to change an index -- to add another column to the key, for example -- you have to drop and re-create it.

Clustering indexes

You can create one index on each table as the clustering index. A clustering index is useful when the table data is often referenced in a particular order. The *clustering index* defines the order in which data is stored in the database. During inserts, DATABASES attempts to place new rows close to rows with similar keys. Then, during queries requiring data in the clustering index sequence, the data can be retrieved faster.

To create an index as the clustering index, specify the CLUSTER clause on the CREATE INDEX statement:

```
CREATE INDEX IAUTHBKNAME ON BOOKS (AUTHORID,BOOKNAME) CLUSTER
```

This statement creates an index on the AUTHORID and BOOKNAME columns as the clustering index. This index would improve the performance of queries written to list authors and all the books that they have written.

Using included columns in indexes

When creating an index, you have the option to include extra column data that is stored with the key, but is not actually part of the key itself and is not sorted. The main reason for including additional columns in an index is to improve the performance of certain queries: with this data already available in the index page, DATABASES does not need to access the data page to fetch it. Included columns can only be defined for unique indexes. However, the included columns are not considered when enforcing uniqueness of the index.

Suppose that you often need to get a list of book names ordered by BOOKID. The query would look like this:

```
SELECT BOOKID,BOOKNAME FROM BOOK ORDER BY BOOKID
```

Create an index that might improve performance:

```
CREATE UNIQUE INDEX IBOOKID ON BOOKS (BOOKID)
INCLUDE(BOOKNAME)
```

As a result, all the data required for the query result is present in the index and no data pages need to be retrieved.

So why not just include all the data in the indexes? First of all, this would require more physical space in the database because the table data would essentially be duplicated in the index. Second, all the copies of the data would need to be updated whenever the data value is updated, and this would be significant overhead in a database where many updates occur.

What indexes should I create?

Consider the following when creating indexes:

- Because indexes are a permanent list of the key values, they require space in the database. Creating many indexes requires additional storage space in your database. The amount of space required is determined by the length of the key columns. DATABASES provides a tool to help you estimate the size of an index.
- Indexes are additional copies of the values so they must be updated if the data in the table is updated. If table data is frequently updated, consider what impact additional indexes have on update performance.

- Indexes significantly improve performance of queries when defined on the appropriate columns.

Creating a view

The `CREATE VIEW` SQL statement is used to define a view. A `SELECT` statement is used to specify which rows and columns are presented in the view.

For example, imagine that you want to create a view that shows only the nonfiction books in our BOOKS table:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
```

After you define this view, there are entries for it in SYSCAT.VIEWS, SYSCAT.VIEWDEP, and SYSCAT.TABLES.

To define column names in the view that are different from those in the base table, you can specify them in the `CREATE VIEW` statement. The following statement creates a MYBOOKVIEW view that contains two columns: TITLE, which represents the BOOKNAME column, and TYPE, which represents the BOOKTYPE column.

```
CREATE VIEW MYBOOKVIEW (TITLE,TYPE) AS
       SELECT BOOKNAME,BOOKTYPE FROM BOOKS
```

The `DROP VIEW` SQL statement is used to drop a view from the database. If you drop a table or another view on which a view is based, the view remains defined in the database but becomes inoperative. The VALID column of SYSCAT.VIEWS indicates whether a view is valid (Y) or not (X). Even if you recreate the base table, the orphaned view remains invalid; you have to recreate it as well.

You can drop the NONFICTIONBOOKS view from the database:

```
DROP VIEW NONFICTIONBOOKS
```

You cannot modify a view; to change a view definition, drop it and recreate it. Use the `ALTER VIEW` statement provided only to modify reference types.

Read-only and updatable views

When you create a view, you can define it as either a *read-only* view or as an *updatable* view. The `SELECT` statement of a view determines whether the view is read-only or updatable. Generally, if the rows of a view can be mapped to rows of the base table, then the view is updatable. For example, the view NONFICTIONBOOKS, as you defined it in the previous example, is updatable because each row in the view is a row in the base table.

The NONFICTIONBOOKS view defined previously includes only the rows where the BOOKTYPE is N. If you insert into the view a row where the BOOKTYPE is F, DATABASES inserts the row into the base table BOOKS. However, if you then select from the view, the newly inserted row cannot be seen through the view. If you do not want to allow a user to insert rows that are outside the scope of the view, you can define the view with the *check option.* Defining a view using `WITH CHECK OPTION` tells DATABASES to check that statements using the view satisfy the conditions of the view.

The following statement defines a view using `WITH CHECK OPTION`:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
       WITH CHECK OPTION
```

This view still restricts the user to seeing only non-fiction books; in addition, it also prevents the user from inserting rows that do not have a value of N in the BOOKTYPE column and updating the value of the BOOKTYPE column in existing rows to a value other than N. The following statements, for instance, are no longer be allowed:

```
INSERT INTO NONFICTIONBOOKS VALUES (...,'F');
UPDATE NONFICTIONBOOKS SET BOOKTYPE = 'F' WHERE BOOKID = 111
```

Nested views with check option

When defining nested views, the check option can be used to restrict operations. However, there are other clauses you can specify to define how the restrictions are inherited. The check option can be defined either as `CASCADED` or `LOCAL`. `CASCADED` is the default if the keyword is not specified. Several possible scenarios explain the differences between the behavior of `CASCADED` and `LOCAL`.

When a view is created `WITH CASCADED CHECK OPTION`, all statements executed against the view must satisfy the conditions of the view and all underlying views -- even if those views were not defined with the check option. Suppose that the view NONFICTIONBOOKS is created without the check option, and you also create a view NONFICTIONBOOKS1 based on the view NONFICTIONBOOKS using the CASCADED keyword:

```
CREATE VIEW NONFICTIONBOOKS AS
```

```
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
CREATE VIEW NONFICTIONBOOKS1 AS
       SELECT * FROM NONFICTIONBOOKS WHERE BOOKID > 100
       WITH CASCADED CHECK OPTION
```

The following `INSERT` statements would not be allowed because they do not satisfy the conditions of at least one of the views:

```
INSERT INTO NONFICTIONBOOKS1 VALUES( 10,..,'N')
INSERT INTO NONFICTIONBOOKS1 VALUES(120,..,'F')
INSERT INTO NONFICTIONBOOKS1 VALUES( 10,..,'F')
```

However, the following `INSERT` statement *would* be allowed because it satisfies the conditions of both of the views:

```
INSERT INTO NONFICTIONBOOKS1 VALUES(120,...,'N')
```

Next, suppose you create a view NONFICTIONBOOKS2 based on the view NONFICTIONBOOKS using `WITH LOCAL CHECK OPTION`. Now, statements executed against the view need only satisfy conditions of views that have the check option specified:

```
CREATE VIEW NONFICTIONBOOKS AS
       SELECT * FROM BOOKS WHERE BOOKTYPE = 'N'
CREATE VIEW NONFICTIONBOOKS2 AS
       SELECT * FROM NONFICTIONBOOKS WHERE BOOKID > 100
       WITH LOCAL CHECK OPTION
```

In this case, the following `INSERT` statements would not be allowed because they do not satisfy the `BOOKID > 100` condition of the NONFICTIONBOOKS2 view:

```
INSERT INTO NONFICTIONBOOKS2 VALUES(10,..,'N')
INSERT INTO NONFICTIONBOOKS2 VALUES(10,..,'F')
```

However, the following `INSERT` statements *would* be allowed even though the value N does not satisfy the `BOOKTYPE = 'N'` condition of the NONFICTIONBOOKS view:

```
INSERT INTO NONFICTIONBOOKS2 VALUES(120,..,'N')
INSERT INTO NONFICTIONBOOKS2 VALUES(120,..,'F')
```

Using joins to retrieve data from more than one table

A *join* is a query that combines data from two or more tables. It is often necessary to select information from two or more tables because required data is often distributed. A join adds columns to the result set. For example, a full join of two three-column tables produces a result set with six columns.

The simplest join is one in which there are no specified conditions. For example:

```
SELECT deptnumb, deptname, manager, id, name, dept, job
  FROM org, staff
```

This statement returns all combinations of rows from the ORG table and the STAFF table. The first three columns come from the ORG tables, and the last four columns come from the STAFF table. Such a result set (the *cross product* of the two tables) is not very useful. What is needed is a *join condition* to refine the result set. For example, here is a query that is designed to identify staff members who are managers:

```
SELECT deptnumb, deptname, id AS manager_id, name AS manager
  FROM org, staff
  WHERE manager = id
  ORDER BY deptnumb
```

And here is a partial result set returned by this query:

Identifying staff members

DEPTNUMB	DEPTNAME	MANAGER_ID	MANAGER
10	Head Office	160	Molinare
15	New England	50	Hanes
20	Mid Atlantic	10	Sanders

The statement you looked at in the last section is an example of an inner join. *Inner joins* return only rows from the cross product that meet the join condition. If a row exists in one table but not the other, it is not included in the result set. To explicitly specify an inner join, rewrite the previous query with an `INNER JOIN` operator in the `FROM` clause:

```
...
  FROM org INNER JOIN staff
  ON manager = id
...
```

The keyword `ON` specifies the join conditions for the tables being joined. DeptNumb and DeptName are columns in the ORG table, and Manager_ID and Manager are based on columns (ID and Name) in the STAFF table. The result set for the inner join consists of rows that have matching values for the Manager and ID columns in the *left table* (ORG)

and the *right table* (STAFF), respectively. (When you perform a join on two tables, you arbitrarily designate one table to be the left table and the other to be the right.)

Outer joins return rows that are generated by an inner join operation, plus rows that would not be returned by the inner join operation. There are three types of outer joins:

- A *left outer join* includes the inner join plus the rows from the *left* table that are not returned by the inner join. This type of join uses the `LEFT OUTER JOIN` (or `LEFT JOIN`) operator in the `FROM` clause.
- A *right outer join* includes the inner join *plus* the rows from the *right* table that are not returned by the inner join. This type of join uses the `RIGHT OUTER JOIN` (or `RIGHT JOIN`) operator in the `FROM` clause.
- A *full outer join* includes the inner join *plus* the rows from *both the left table and the right table* that are not returned by the inner join. This type of join uses the `FULL OUTER JOIN` (or `FULL JOIN`) operator in the `FROM` clause.

Construct more complex queries to answer more difficult questions. The following query is designed to generate a list of employees who are responsible for projects, identifying those employees who are also managers by listing the departments that they manage:

```
SELECT empno, deptname, projname
  FROM (employee
  LEFT OUTER JOIN project
  ON respemp = empno)
  LEFT OUTER JOIN department
  ON mgrno = empno
```

The first outer join gets the name of any project for which the employee is responsible; this outer join is enclosed by parentheses and is resolved first. The second outer join gets the name of the employee's department if that employee is a manager.

Using set operators to combine two or more queries into a single query

Combine two or more queries into a single query by using the `UNION`, `EXCEPT`, or `INTERSECT` set operators. *Set operators* process the results of the queries, eliminate duplicates, and return the final result set.

- The `UNION` set operator generates a result table by combining two or more other result tables.
- The `EXCEPT` set operator generates a result table by including all rows that are returned by the first query, but not by the second or any subsequent queries.
- The `INTERSECT` set operator generates a result table by including only rows that are returned by all the queries.

Following is an example of a query that makes use of the UNION set operator. The same query could use the EXCEPT or the INTERSECT set operator by substituting the appropriate keyword for UNION.

```
"SELECT sales_person FROM sales
  WHERE region = 'Ontario-South'
UNION
SELECT sales_person FROM sales
  WHERE sales > 3"
```

Using the GROUP BY clause to summarize results

Use the GROUP BY clause to organize rows in a result set. Each group is represented by a single row in the result set. For example:

```
SELECT sales_date, MAX(sales) AS max_sales FROM sales
  GROUP BY sales_date
```

This statement returns a list of sales dates from the SALES table. The SALES table in the SAMPLE database contains sales data, including the number of successful transactions by a particular sales person on a particular date. There is typically more than one record per date. The GROUP BY clause groups the data by sales date, and the MAX function in this example returns the maximum number of sales recorded for each sales date.

A different flavor of the GROUP BY clause includes the specification of the GROUPING SETS clause. *Grouping sets* can be used to analyze data at different levels of aggregation in a single pass. For example:

```
SELECT YEAR(sales_date) AS year, region, SUM(sales) AS tot_sales
  FROM sales
  GROUP BY GROUPING SETS (YEAR(sales_date), region, () )
```

Here, the YEAR function is used to return the year portion of date values, and the SUM function is used to return the total in each set of grouped sales figures. The *grouping sets list* specifies how the data is to be grouped, or *aggregated.* A pair of empty parentheses is added to the grouping sets list to get a grand total in the result set. The statement returns the following:

Result of Grouping Sets command

YEAR	REGION	TOT_SALES
-	-	155
-	Manitoba	41
-	Ontario-North	9
-	Ontario-South	52

-	Quebec	53
1995	-	8
1996	-	147

A statement that is almost identical to the previous one, but that specifies the `ROLLUP` clause, or the `CUBE` clause instead of the `GROUPING SETS` clause, returns a result set that provides a more detailed perspective on the data. It might provide summaries by location or time.

The `HAVING` clause is often used with a `GROUP BY` clause to retrieve results for groups that satisfy only a specific condition. A `HAVING` clause can contain one or more predicates that compare some property of the group with another property of the group or a constant. For example:

```
"SELECT sales_person, SUM(sales) AS total_sales FROM sales
  GROUP BY sales_person
  HAVING SUM(sales) > 25"
```

This statement returns a list of salespeople whose sales totals exceed 25.

The COMMIT and ROLLBACK statements and transaction boundaries

Units of work and savepoints

A *unit of work* (UOW), also known as a *transaction*, is a *recoverable* sequence of operations within an application process. The classic example of a UOW is a simple bank transaction to transfer funds from one account to another. There is an inconsistency -- immediately after the application subtracts an amount of money from one account -- that is resolved after an equal amount of money is added to the second account. When these changes have been committed, they become available to other applications.

A UOW starts implicitly when the first SQL statement within an application process is issued against the database. All subsequent reads and writes by the same application process are considered part of the same UOW. The application ends the UOW by issuing either a `COMMIT` or a `ROLLBACK` statement, whichever is appropriate. The `COMMIT` statement makes all changes made within the UOW permanent, whereas the `ROLLBACK` statement reverses those changes. If the application ends normally without an explicit `COMMIT` or `ROLLBACK` statement, the UOW is automatically committed. If the application ends abnormally before the end of a UOW, that unit of work is automatically rolled back.

A *savepoint* lets you selectively roll back a subset of actions that make up a UOW without losing the entire transaction. You can nest savepoints and have several *savepoint levels* active at the same time; this allows your application to roll back to a specific savepoint, as necessary. Suppose you have three savepoints (A, B, and C) defined within a particular UOW:

```
do some work;
 savepoint A;
 do some more work;
  savepoint B;
  do even more work;
   savepoint C;
   wrap it up;
  roll back to savepoint B;
```

The rollback to savepoint B automatically releases savepoint C, but savepoints A and B remain active.

Creating and calling an SQL procedure

An SQL *procedure* is a stored procedure whose body is written in SQL. The body contains the logic of the SQL procedure. It can include variable declarations, condition handling, flow-of-control statements, and DML. Multiple SQL statements can be specified within a *compound statement*, which groups several statements together into an executable block.

An SQL procedure is created when you successfully invoke a `CREATE PROCEDURE (SQL)` statement, which defines the SQL procedure with an application server. SQL procedures are a handy way to define more complex queries or tasks that can be called whenever they are needed.

An easy way to create an SQL procedure is to code the `CREATE PROCEDURE (SQL)` statement in a command line processor (CLP) script. For example, if the statement shown below were in a file called `createSQLproc.Databases`, that file could be executed to create the SQL procedure:

1. Connect to the SAMPLE database.
2. Issue the following command:

   ```
   Databases -td@ -vf createSQLproc.Databases
   ```

This `Databases` command specifies the `-td` option flag, which tells the command line processor to define and to use `@` as the statement termination character (because the semicolon is already being used as a statement termination character inside the procedure body); the `-v` option flag, which tells the command line processor to echo command text to standard output; and the `-f` option flag, which tells the command line processor to read command input from the specified file instead of from standard input.

```
CREATE PROCEDURE sales_status
(IN quota INTEGER, OUT sql_state CHAR(5))
BEGIN
```

```
  DECLARE SQLSTATE CHAR(5);
  DECLARE rs CURSOR WITH RETURN FOR
  SELECT sales_person, SUM(sales) AS total_sales
    FROM sales
    GROUP BY sales_person
    HAVING SUM(sales) > quota;
  OPEN rs;
  SET sql_state = SQLSTATE;
END @
```

This procedure, called SALES_STATUS, accepts an input parameter called *quota* and returns an output parameter called *sql_state.* The procedure body consists of a single `SELECT` statement that returns the name and the total sales figures for each salesperson whose total sales exceed the specified quota.

Most SQL procedures accept at least one input parameter. In our example, the input parameter contains a value (*quota*) that is used in the `SELECT` statement contained in the procedure body.

Many SQL procedures return at least one output parameter. Our example includes an output parameter (*sql_state*) that is used to report the success or failure of the SQL procedure. DATABASES returns an `SQLSTATE` value in response to conditions that could be the result of an SQL statement. Because the returned `SQLCODE` or `SQLSTATE` value pertains to the last SQL statement issued in the procedure body, and accessing the values alters the subsequent values of these variables (because an SQL statement is used to access them), the `SQLCODE` or `SQLSTATE` value should be assigned to and returned through a locally defined variable (such as the *sql_state* variable in our example).

The parameter list for an SQL procedure can specify zero or more parameters, each of which can be one of three possible types:

- `IN` parameters pass an input value to an SQL procedure; this value cannot be modified within the body of the procedure.
- `OUT` parameters return an output value from an SQL procedure.
- `INOUT` parameters pass an input value to an SQL procedure and return an output value from the procedure.

SQL procedures can return zero or more result sets. In our example, the SALES_STATUS procedure returns one result set. This has been done by:

1. Declaring the number of result sets that the SQL procedure returns in the `DYNAMIC RESULT SETS` clause.
2. Declaring a cursor in the procedure body (using the `WITH RETURN FOR` clause) for each result set that is returned. A *cursor* is a named control structure that is used by an application program to point to a specific row within an ordered set of rows. A cursor is used to retrieve rows from a set.
3. Opening the cursor for each result set that is returned.

4. Leaving the cursor(s) open when the SQL procedure returns.

Variables must be declared at the beginning of the SQL procedure body. To *declare* a variable, assign a unique identifier to and specify an SQL data type for the variable and, optionally, assign an initial value to the variable.

The `SET` clause in our sample SQL procedure is an example of a *flow-of-control* clause. The following flow-of-control statements, structures, and clauses can be used for conditional processing within an SQL procedure body:

- The `CASE` structure selects an execution path based on the evaluation of one or more conditions.
- The `FOR` structure executes a block of code for each row of a table.
- The `GET DIAGNOSTICS` statement returns information about the previous SQL statement into an SQL variable.
- The `GOTO` statement transfers control to a labeled block (a section of one or more statements identified by a unique SQL name followed by a colon).
- The `IF` structure selects an execution path based on the evaluation of conditions. The `ELSEIF` and `ELSE` clauses enable you to branch or to specify a default action if the other conditions are false.
- The `ITERATE` clause passes the flow of control to the beginning of a labeled loop.
- The `LEAVE` clause transfers program control out of a loop or block of code.
- The `LOOP` clause executes a block of code multiple times until a `LEAVE`, `ITERATE`, or `GOTO` statement transfers control outside of the loop.
- The `REPEAT` clause executes a block of code until a specified search condition returns true.
- The `RETURN` clause returns control from the SQL procedure to the caller.
- The `SET` clause assigns a value to an output parameter or SQL variable.
- The `WHILE` clause repeatedly executes a block of code while a specified condition is true.

To successfully create SQL procedures, you must have installed the DATABASES Application Development Client on the database server. (See the first tutorial in this series for more on the Application Development Client.) The dependency on a C compiler to create SQL procedures was eliminated in DATABASES Universal Database Version 8. All of the operations that were dependent on a C compiler are now performed by DATABASES-generated byte code that is hosted in a virtual machine. For more information about this enhancement, see Resources.

Use the SQL `CALL` statement to call SQL procedures from the DATABASES command line. The procedure being called must be defined in the system catalog. Client applications written in any supported language can call SQL procedures. To call the SQL procedure SALES_STATUS, perform the following steps:

1. Connect to the SAMPLE database.
2. Issue the following statement:

```
Databases CALL sales_status (25, @)
```

Because parentheses have special meaning to the command shell on UNIX-based systems, on those systems they must be preceded with a backslash (\) character or be enclosed by double quotation marks:

```
Databases "CALL sales_status (25, @)"
```

Do not include double quotation marks if you are using the command line processor (CLP) in interactive input mode, characterized by the `Databases =>` input prompt.

In this example, a value of 25 for the input parameter *quota* is passed to the SQL procedure, as well as a question mark (?) place-holder for the output parameter *sql_state*. The procedure returns the name and the total sales figures for each salesperson whose total sales exceed the specified quota (25). The following is sample output returned by this statement:

```
SQL_STATE: 00000
SALES_PERSON            TOTAL_SALES
GOUNOT                       50
LEE                          91
 "SALES_STATUS" RETURN_STATUS: "0"
```

Examples :

Create Table :

```
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)


CREATE TABLE EMPLOYEE
   (ID         SMALLINT NOT NULL,
    NAME       VARCHAR(9),
    DEPT       SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
    JOB        CHAR(5) CHECK (JOB IN ('Sales','Mgr','Clerk')),
    HIREDATE    DATE,
    SALARY     DECIMAL(7,2),
    COMM       DECIMAL(7,2),
```

```
 PRIMARY KEY (ID),
 CONSTRAINT  YEARSAL CHECK (YEAR(HIREDATE) > 1986
  OR SALARY > 40500)
)
```

Create View :

```
CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2

CREATE VIEW MA_PROJ
    AS SELECTPROJNO, PROJNAME, RESPEMP
    FROM PROJECT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'

 CREATE VIEW PRJ_LEADER
    AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
    FROM PROJECT, EMPLOYEE
    WHERE RESPEMP = EMPNO


 CREATE VIEW PRJ_LEADER
    AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP,
              LASTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
      FROM PROJECT, EMPLOYEE
      WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

Create Index :

```
CREATE INDEX CORPDATA.INX1 ON
CORPDATA.EMPLOYEE (LASTNAME)

CREATE INDEX UNIQUE_NAM
     ON PROJECT(PROJNAME)

CREATE UNIQUE INDEX JOB_BY_DEPT
    ON EMPLOYEE (WORKDEPT, JOB)


CREATE INDEX IDX1 ON TAB1 (col1) COLLECT STATISTICS
```

Stored Procedures:

Sends out the procedure language defined in the sysprocedures table

```
CREATE PROCEDURE OUT_LANGUAGE (OUT procedureLanguage CHAR(8))
DYNAMIC RESULT SETS 0
LANGUAGE SQL
READS SQL DATA
BEGIN
  SELECT language INTO procedureLanguage
    FROM sysibm.sysprocedures
    WHERE procname = 'OUT_LANGUAGE';
END
/
```

Updates Salary for employee number and rate sent in:

```
CREATE OR REPLACE PROCEDURE UPDATE_SALARY
(IN EMPLOYEE_NUMBER CHAR(10),
IN RATE DECIMAL(6,2))
LANGUAGE SQL
IF RATE <= 0.50
THEN UPDATE EMP
SET SALARY = SALARY * RATE
WHERE EMPNO = EMPLOYEE_NUMBER;
ELSE UPDATE EMP
SET SALARY = SALARY * 0.50
WHERE EMPNO = EMPLOYEE_NUMBER;
END IF
/
```