This tutorial discusses IBM® DB2® 10.1 support for data types, tables, views, triggers, constraints and indexes. It explains the features of these objects, how to create and manipulate them using Structured Query Language (SQL), and how they can be used in an application. This tutorial is the fifth in a series that you can use to help prepare for the DB2 10.1 Fundamentals certification exam 610.

View more content in this series

Before you start
Learn what to expect from this tutorial, and how to get the most out of it.

About this series
Thinking about obtaining the IBM Certified Database Associate — DB2 10.1 Fundamentals certification? If so, you've come to the right place. This DB2 10.1 Fundamentals certification preparation series is designed to cover all the topics you need to know before you sit down to take the DB2 10.1 Fundamentals certification exam (Exam 610). Even if you're not planning to seek certification right away, the information presented in this series can help you learn about many of the features and functionality available in DB2 10 for z/OS® and DB2 10.1 for Linux®, UNIX®, and Windows®.

You don't see the tutorial you're looking for yet? You can review the DB2 9 tutorials in the DB2 9 Fundamentals certification 730 prep series.

About this tutorial
Twenty-two percent of the DB2 10.1 Fundamentals certification exam (Exam 610) is designed to test your knowledge of the structure, creation, and alterations of tables, views, and indexes within DB2. This includes the data types that are supported, the various forms of tables and views, the restrictions that can be created on the values to be entered into tables and views, the indexes
that can be built to provide speedy access, and triggers that can be built to cause other actions to happen upon certain events.

**Objectives**

The material in this tutorial covers the objectives in Section 5 of the DB2 10.1 Fundamentals certification exam (Exam 610). You can view these objectives at [http://www-03.ibm.com/certify/tests/obj610.shtml](http://www-03.ibm.com/certify/tests/obj610.shtml).

After completing this tutorial, you should be able to:

- Have the ability to demonstrate usage of DB2 data types (XML data types, Oracle compatibility data types).
- Have the ability to create a temporary table.
- Have the knowledge to identify when referential integrity should be used.
- Have the knowledge to identify methods of data constraint.
- Have the knowledge to identify characteristics of a table, view or index.
- Have the knowledge to identify when triggers should be used.
- Have the knowledge of schemas.

**Prerequisites**

In order to understand some of the material presented here, you should be familiar with the following notions:

- **Database Management System (DBMS)** — a system that provides the core services necessary to support collecting, maintaining, protecting, sharing, and retrieving data among authorized users and applications. Generally it uses the devices and communications services made available by an operating system, and is used by applications which implement business logic and control the interactions with the users of the system.
- **Relational Database Management System** — a database management system that is perceived by the user as named tables in which data is arranged in rows and named columns.
- **Structured Query Language (SQL)** — a standardized language used to define objects and manipulate data in a relational database management system.
- **Special register** — a storage area that is defined for an application process by DB2 and is used to store information that can be referenced in SQL statements. Examples of a "special register" are CURRENT TIMESTAMP or CURRENT USER.

**System requirements**

You do not need a copy of DB2 to complete this tutorial, but if you have access to a DB2 database server, you will be able to test some of the commands and concepts presented.

You can download a complimentary copy of DB2 Express-C from IBM.

**Schemas**

A schema is a collection of named objects such as tables, views, triggers, functions, procedures, or indexes. It provides a way to group those objects logically via a naming convention. It also
provides a way to use the same natural name for several objects, and to prevent ambiguous references to those objects.

For example, the schema names 'APP1' and 'APP2' make it easy to distinguish two different SALES tables, APP1.SALES and APP2.SALES, which do not need to have the same structure.

A DB2 schema is distinct from, and should not be confused with, an XML schema, which is a standard that describes the structure and validates the content of XML documents.

If an object is specifically qualified with a schema name when created, the object is assigned to that schema. If no schema name is specified when the object is created, the default schema name is used, as specified in the CURRENT SCHEMA special register. SET CURRENT SCHEMA='MYSCHEMA';

Tables

Data in DB2 is contained in tables. A table consists of one or more columns of various data types. The data is structured into rows, each with the structure specified for the table.

Tables are defined using the CREATE TABLE SQL statement.

DB2 supports the following types of tables.

- The system catalog tables.
- Base tables, which contain persistent data and are directly queryable.
- Temporal tables:
  - System period temporal tables: to keep a history of all values of all rows.
  - Business period temporal tables: to allow application-defined periods of effectivity for each row.
  - Bi-temporal tables: to incorporate both system period and business period features.
- Typed tables: (Note: Linux, UNIX, and Windows only) contain data as defined by a user-defined structured data type.
- Materialized query tables: to contain data derived from massive data, directly queryable.
- Temporary tables:
  - Created: the definition is cataloged and shared with all users, but each session can easily obtain its own instance, with no data shared with other sessions.
  - Declared: the definition is not shared nor is the data.
- Clone tables: (Note: z/OS only) to allow maintenance of high availability, high performance tables, not directly queryable.
- Columnar data tables: (Note: z/OS only), not directly queryable.
  - Auxiliary tables: for various LOB column data.
  - XML tables: for XML column data.

Each DB2 server (referred to on Linux, UNIX, and Windows as a "database", and on z/OS as a "subsystem") has a set of tables, called the system catalog tables, which hold information about the objects within the server.
Base tables
Use the `CREATE TABLE` SQL statement to define a table. The statement shown in Listing 1 creates a simple table named BOOKS that contains three columns:

**Listing 1. Simple table with three columns**

```sql
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:

**Listing 2. Using CREATE TABLE**

```sql
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 identity 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 (see the Related topics section).

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. The `LOAD` utility, intended for loading large volumes of data, places rows directly onto data pages, maintains indexes in bulk rather than row-by-row, and has the capability to avoid logging. Note: Linux, UNIX, and Windows provides an `IMPORT` utility to insert rows from a file of data. It uses `INSERT` statements, and is designed for loading small amounts of data.

Storing tables
Tables are stored in tablespaces. Tablespaces have physical space allocated to them.

When you create a table, you can let DB2 place the table in a default tablespace, or specify the tablespace in which you'd like the table to reside, as shown in Listing 3.

**Listing 3. Default or specify tablespace**

```sql
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 the appropriate definition of tablespaces can have significant effects upon the performance and maintainability of the database. For more information on tablespaces, check out the second tutorial in this series.

**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 statement shown in Listing 4 adds a column called `BOOKTYPE` to the `BOOKS` table.

**Listing 4. Add BOOKTYPE column**

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

You can also change the following characteristics of specific columns in a table.

- The identity attributes of a column.
- The length of a string column.
- The data type of a column.
- The null ability of a column.
- The constraint of a column.

There are the following restrictions on altering columns.

- When altering the data type of a column, the new data type 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 data type 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.
- When altering a string column, you cannot decrease the length.

The statement shown in Listing 5 changes the data type of column `BOOKNAME` from VARCHAR(100) to VARCHAR(200), and also changes the null ability of the ISBN column to NOT NULL WITH DEFAULT 'UNKNOWN':

**Listing 5. Change data type of column**

```sql
ALTER TABLE BOOKS ALTER BOOKNAME SET DATA TYPE VARCHAR(200) ALTER ISBN SET DEFAULT 'UNKNOWN';
```
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 data type of a column to an incompatible data type (see the section on Data types). To change characteristics such as these, save the table data, drop the table, and then 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 DROP TABLE statement, as shown in Listing 6, deletes the BOOKS table and its data:

**Listing 6. Delete table and data**

```sql
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, then specify the NOT NULL clause for the column. Specify a default value using the WITH DEFAULT clause and a default value. Note that if a column allows nulls, that column will require one extra byte to hold the null indicator. Any null value is not equal to, greater than, or less than to any actual value, nor is it equal to any other null value other than the special value “NULL” which can be used in SQL.

The CREATE TABLE statement, as shown in Listing 7, creates a table called BOOKS, where the BOOKID column does not allow null values and the default value for BOOKNAME is the string with value TBD.

**Listing 7. Using CREATE TABLE statement**

```sql
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 DB2 is to generate a BOOKID using the GENERATED ALWAYS AS IDENTITY clause, as shown in Listing 8.

**Listing 8. Using the GENERATED ALWAYS AS IDENTITY clause**

```sql
CREATE TABLE BOOKS ( BOOKID INTEGER NOT NULL,
                      GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
                      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.
Note: In DB2 for Linux, UNIX, and Windows, it is also possible to use the GENERATED ALWAYS option to have DB2 calculate the value of a column derived from other columns of the same row automatically.

Temporal Tables

DB2 provides technology to enable tracking and querying of historical, current, and planned conditions in a straightforward and efficient manner via direct support for temporal tables.

There are two types of temporal tables: system period and business period.

In a system period temporal table, images of rows are retained in an associated history table. The timestamps are maintained by the system and reflect the times during which each row contained the values recorded. This allows a complete historical perspective of the contents of a table. One can query to obtain the results that would have been produced at any specific time in the past.

In a business period temporal table, the meaning of the timestamp is determined by the business, and its timestamps are under application control. Most commonly, these will reflect the timestamp when the business intends (or intended) certain conditions to be effective, such as when a given interest rate will take effect, or when a certain insurance policy was in force. This type of table allows one to query past, present or future business conditions.

These two types of temporal tables are not exclusive. DB2 allows you to define a table with both types of temporal attributes. These are referred to as bi-temporal tables.

Creating a table with PERIOD SYSTEM_TIME is a three step process.

1. Create the base table for current data only, but include three TIMESTAMP columns: two for start/end points of the system time, and one for the transaction start time (used internally), and specify PERIOD SYSTEM_TIME. These may be added to an existing table by an ALTER. Listing 9 shows a typical CREATE TABLE statement containing these fields.

   **Listing 9. Creating base table for current data**

   ```sql
   CREATE TABLE POLICY ( 
   POLICYID INTEGER NOT NULL PRIMARY KEY, 
   VIN VARCHAR(10) NOT NULL, 
   COVERAGE INTEGER NOT NULL, 
   SYS_START TIMESTAMP(12) NOT NULL 
   GENERATED ALWAYS AS ROW BEGIN, 
   SYS_END TIMESTAMP(12) NOT NULL 
   GENERATED ALWAYS AS ROW END, 
   TRAN_START TIMESTAMP(12) 
   GENERATED ALWAYS AS TRANSACTION START ID, 
   PERIOD SYSTEM_TIME (SYS_START, SYS_END) );
   ```

2. Create an identically structured history table, easily done using the CREATE TABLE ... LIKE syntax, as shown in Listing 10.

   **Listing 10. Use CREATE TABLE ... LIKE syntax**

   ```sql
   CREATE TABLE POLICY HISTORY LIKE POLICY;
   ```

3. Alter the base table to activate the history table, as shown in Listing 11.
Listing 11. Alter base table

```
ALTER TABLE POLICY ADD VERSIONING USE HISTORY TABLE POLICY_HISTORY;
```

Insert, update, and delete data to the base table as normal, but do not provide values for the three timestamps (as these are always generated), as shown in Listing 12.

Listing 12. Insert, update, and delete data

```
INSERT INTO POLICY(POLICYID, VIN, COVERAGE)
VALUES(1001, '11235811', 100000);
COMMIT;
UPDATE POLICY SET COVERAGE=200000 WHERE POLICYID=1001;
COMMIT;
DELETE FROM POLICY WHERE POLICYID=1001;
COMMIT;
```

The old versions of the rows are copied into POLICY_HISTORY automatically as part of the changes to the values in the POLICY table, with the appropriate system timestamps. At the conclusion of the previous three steps (if they were the only actions performed and each was committed) there would be no rows in the POLICY table, but two separate rows in the POLICY_HISTORY table showing the two sets of values and the timestamps indicating when they had these values.

The query shown in Listing 13 can be used to find out what policy was in effect at 6pm on December 1, 2011. Note that this "time-travelling" query refers only to the POLICY table, not explicitly to the POLICY_HISTORY. For more on queries and "time-travelling" queries, see Section 4 - Working with DB2 data using SQL.

Listing 13. Find policy

```
SELECT COVERAGE
FROM POLICY FOR SYSTEM_TIME AS OF '2011-12-01-18.00.00'
WHERE POLICYID = 1001;
```

Creating a table with PERIOD BUSINESS_TIME is just a one-step process, since no separate history table is involved. Just include the two start/end timestamps (or dates) to be maintained by the application, and the PERIOD BUSINESS_TIME clause, as shown in Listing 14.

Listing 14. Creating table with PERIOD BUSINESS_TIME

```
CREATE TABLE POLICY (
    POLICYID INTEGER NOT NULL,
    VIN VARCHAR(10) NOT NULL,
    COVERAGE INTEGER NOT NULL,
    POLICY_START DATE NOT NULL,
    POLICY_END DATE NOT NULL,
    PERIOD BUSINESS_TIME (POLICY_START, POLICY_END),
    PRIMARY KEY(POLICYID, BUSINESS_TIME WITHOUT OVERLAPS) );
```

With the WITHOUT OVERLAPS clause, DB2 will ensure that the rows for a given POLICYID will not overlap periods.
Materialized query tables

Materialized query tables contain data that is derived from one or more source tables, according to its defining SQL query, much like a view. Unlike a view though, the data is physically stored. The data can be refreshed according to a system schedule (system-maintained) or user direction (user-maintained).

Materialized query tables are useful for complex queries that run on large amounts of data. DB2 can pre-compute all or part of such queries and use the pre-computed or materialized results to answer the queries more efficiently. Materialized query tables are commonly used in data warehousing and business intelligence applications.

Only user-maintained materialized query tables can also be updated with the LOAD utility and the UPDATE, INSERT, and DELETE SQL statements.

Materialized query tables can be used to improve the performance of dynamic SQL queries. If DB2 determines that a portion of a query could be resolved using a materialized query table, the query might be rewritten by DB2 to use the materialized query table.

A materialized query table is created with a `CREATE TABLE` statement using a SQL full select and some other parameters to control its refresh and availability for query optimization.

The DDL in Listing 15 creates a materialized query table to contain a summary of the current inventory of BOOKS to be refreshed when directed by the user.

Listing 15. Creating a materialized query table refreshed by the user

```
CREATE TABLE INVENTORY(BOOKTYPE, AUTHORS, TITLES, BOOKS)
AS (SELECT BOOKTYPE, COUNT(DISTINCT AUTHORID), COUNT(DISTINCT ISBN),
    COUNT(DISTINCT BOOKID)
    FROM BOOKS
    GROUP BY BOOKTYPE)
DATA INITIALLY DEFERRED REFRESH DEFERRED
MAINTAINED BY USER;
```

The data within the table is replaced at any time by issuing (either interactively or from an application program) the SQL statement in Listing 16.

Listing 16. Refresh the data in a materialized query table

```
REFRESH TABLE INVENTORY;
```

Global temporary tables

The `CREATE GLOBAL TEMPORARY TABLE` statement creates a description of a temporary table. Each session that selects from a created temporary table retrieves only rows that the same session has inserted. When the session terminates, the rows of the table associated with the session are deleted. This allows the session to act upon its data with SQL without incurring significant overhead.
The rest of the specification of a created global temporary table is much like the specification of a base table, as is illustrated in Listing 17.

### Listing 17. Create a created global temporary table

```sql
SET CURRENT SQLID='USER1';
CREATE GLOBAL TEMPORARY TABLE LIBRARY.TOBECATALOGED (
    BOOKNAME VARCHAR(100),
    AUTHORNAME VARCHAR(100)
);
GRANT ALL ON LIBRARY.TOBECATALOGED TO USER2, USER3, USER4;
```

Only USER2, USER3 and USER4 are able to use a temporary instance of LIBRARY.TOBECATALOGED. When USER2 mentions LIBRARY.TOBECATALOGED, that instance comes into existence and is accessible only to USER2 from within USER2’s current session. That instance and its data only persists until USER2 issues COMMIT (which is the default). Note too that the GRANT by USER1 must be GRANT ALL. USER2 does not have to create the table instance explicitly, just use it. An example is shown in Listing 18.

### Listing 18. Implicit creation of an instance of a created global temporary table

```sql
SET CURRENT SQLID='USER2';
INSERT INTO LIBRARY.TOBECATALOGED VALUES('This Book Title', 'Ann Author');
INSERT INTO LIBRARY.TOBECATALOGED VALUES('Remember This Also', 'Another Author');
SELECT COUNT(*) FROM LIBRARY.TOBECATALOGED;
COMMIT;
SELECT COUNT(*) FROM LIBRARY.TOBECATALOGED;
```

All instances of LIBRARY.TOBECATALOGED will have the same structure which is defined in the system catalogs. The first SELECT in Listing 18 will return the value 2. The second SELECT will return the value 0, since the definition of LIBRARY.TOBECATALOGED accepted the default that the data will persist only until COMMIT.

The second form of a global temporary table is a declared global temporary table. The table is available only for the session (as was the case with the created global temporary table), and the data put into it is only available within the single session until a COMMIT is issued.

The major difference between declared and created global temporary tables is that the structure of a declared global temporary table is not predefined but rather defined within the same session. Two different sessions can each have their own version (with differing structures) of a table with exactly the same name as the other. Listing 19 illustrates the creation and use of a declared global temporary table.

### Listing 19. Creating and using a declared global temporary table

```sql
SET CURRENT SQLID='USER2';
DECLARE GLOBAL TEMPORARY TABLE TOBECATALOGED (
    BOOKNAME VARCHAR(100),
    AUTHORNAME VARCHAR(100)
);
INSERT INTO SESSION.TOBECATALOGED VALUES('A New Book', 'A Different Author');
SELECT COUNT(*) FROM SESSION.TOBECATALOGED;
COMMIT;
SELECT COUNT(*) FROM SESSION.TOBECATALOGED;
```
The first SELECT in Listing 19 will retrieve the value 1. The second SELECT will retrieve the value 0, since the default for both types of global temporary table is that the data is deleted when a commit is issued.

Note that the schema of any and all declared global temporary tables will be 'SESSION'. The structure of a declared global temporary table is not registered in the system catalogs, so the same table name can be used within several different sessions, each with a different structure.

Note also that all subsequent references to the declared global temporary table should include the schema 'SESSION', since if omitted, DB2 will use the value of the special register CURRENT SQLID to resolve the full name of the table.

**Data types**

DB2 provides a rich and flexible assortment of data types. DB2 comes 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, or more precisely defined types based on built-in data types. Choosing which data 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. Note: on z/OS there is a ROWID data type, and on Linux, UNIX, and Windows there is BOOLEAN.

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

**Numeric data types**

There are four categories of numeric data types: integer, decimal, floating point, and decfloat. 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 up to 31 digits. 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 can be defined as REAL. REAL can be defined with a precision up to 24 digits and requires 4 bytes of storage. DOUBLE can be defined with a precision up to 53 digits and requires 8 bytes of storage. FLOAT can be used as a synonym for REAL or DOUBLE depending upon the specified precision.
• Decfloat: DECFLOAT implements the IEEE standard 754r for floating point, with a precision of 16 or 34 digits. DECFLOAT can occupy either 8 or 16 bytes, depending upon the precision.

String data types
DB2 provides several data types for storing character data or strings. 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 (Note: z/OS, 255 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 approximately 32K bytes. In the database, VARCHAR data only takes as much space as required, plus 2 bytes for the length.

The following data types are used to store strings made of multi-byte characters.

• GRAPHIC is used to store fixed-length multi-byte character strings. The maximum length of a GRAPHIC column is 127 characters.
• VARGRAPHIC is used to store variable-length multi-byte character strings. The maximum length of a VARGRAPHIC column is approximately 16K characters.

DB2 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-1 in length. However, only the space required is actually used. The long data types are as follows.

• CLOB (character large object).
• DBCLOB (double-byte character large object).
• BLOB (binary large object) a LOB for which no character code conversion should be applied, for example, a picture.

Datetime data types
DB2 provides the following 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.
DB2 provides 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. DB2 also provides 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 you are 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.

The string representation of a timestamp is yyyy-mm-dd-hh.mm.ss.nnnnnn.

**XML data type**

DB2 provides the XML data type to store well-formed XML documents. Values in XML columns are stored in an internal representation different from string data types. To store XML data in an XML data type column, transform the data using the XMLPARSE function. An XML data type value can be transformed into a serialized string value representing of the XML document using the XMLSERIALIZE function. DB2 provides many other built-in functions to manipulate XML data types.

**User-defined data types**

DB2 allows you to define data types that suit your application.

User-defined distinct: Define a new data type based on a built-in type. This new type has the same features of the built-in type, but you can use it to ensure that only values of the same type are compared. For example, you can define a Canadian dollar type (CANDOL) and a US dollar type (USADOL), both based on DECIMAL(10,2). Both types are based on the same built-in type, but you won't be able to compare them unless a conversion function is applied. The following CREATE TYPE statements create the CANDOL and USADOL UDTs, as shown in Listing 20.

**Listing 20. CREATE TYPE statements**

```
CREATE DISTINCT TYPE CANDOL AS DECIMAL(10,2) WITH COMPARISONS;
CREATE DISTINCT TYPE USADOL AS DECIMAL(10,2) WITH COMPARISONS;
```

DB2 automatically generates functions to perform casting between the base type and the distinct type, and comparison operators for comparing instances of the distinct type. The following statements show how to create a table with a column of CANDOL type and then insert data into the table using the CANDOL casting function, as shown in Listing 21.

**Listing 21. CANDOL casting function**

```
CREATE TABLE ITEMS (ITEMID CHAR(5), PRICE CANDOL);
INSERT INTO ITEMS VALUES('ABC11',CANDOL(30.50) );
```

In addition, on the Linux, UNIX, and Windows platform you can create the following.
- User-defined array: An array type is defined as an array with elements of another data type.
- User-defined cursor: A cursor type represents a reference to an underlying cursor.
- User defined row: A row type is a structure that can be used by SQL PL as a data type for variables and parameters to represent a row for simpler manipulation. It can also be used as the data type of an array.
- User-defined structured: A structured type has a structure that contains a sequence of attributes each of which has a data type. A structured type may be used as the type of a table, view or column. For example, you can create a structured type named ADDRESS that contains data for street number, street name, city, and so forth. Then you can use this type when defining other tables, such as employees or suppliers, because the same data is required for both. Also, structured types can have subtypes in a hierarchical structure. This lets you store objects that belong to a hierarchy.
- User-defined reference: When using structured types, you can define references to rows in another table using reference types. These references appear similar to referential constraints; however, they do not enforce relationships between the tables. References in tables let you specify queries in a different way.

**DB2 Extenders**

DB2 Extenders provide support for complex, nontraditional data types. They are packaged separately from the DB2 server code and installed in each DB2 instance that uses the data types and methods provided by the extenders.

DB2 Extenders available from IBM and from independent software vendors. The DB2 Spatial Extender can be used for storing and analyzing spatial data. The DB2 Net Search Extender offers users and application programmers a way to search full-text documents stored in DB2 databases, other databases, and file systems using SQL queries.

DB2 Extenders are implemented using the features of user-defined types and user-defined functions (UDFs). Each extender comes with one or more UDTs, UDFs for operating on the UDT, specific application programming interfaces (APIs), and perhaps other tools.

**Constraints**

DB2 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.

DB2 provides three types of constraints: unique, referential integrity, and check constraints.

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. Generally it is good practice to define a primary key on each table if possible. In the previous example, the BOOKS table has a BOOKID column which is used to uniquely identify a book, and so is defined as the primary key. This value is also used in other tables that contain information related to this book. DB2 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 used in other tables. In this case, the ISBN column can be defined as UNIQUE, as shown in Listing 22.

### Listing 22. Defining a UNIQUE key

```sql
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.

DB2 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, DB2 creates a unique index to enforce uniqueness on the column. DB2 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, as shown in Listing 23.

### Listing 23. Failed statements due to PRIMARY KEY

```sql
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, where each book has an author and that author must exist in the AUTHOR table. Each author has a unique identifier stored in the AUTHORTID column. The AUTHORTID 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, as shown in Listing 24.

**Listing 24. Establishing a 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 AUTHOR has a primary key that is mentioned within another table called BOOKS. AUTHOR is called the parent table in the relationship, and BOOKS the dependent table in the relationship.

The constraint can be added when the dependent table is created, or the dependent table can be altered to add the constraint later.

You can also define relationships between rows of the same table. In such a case, the parent table of the relationship and dependent table are the same, as shown in Listing 25.

**Listing 25. Define relationships between rows**

```
CREATE TABLE EMPLOYEE (
    PERSONID CHAR(10) NOT NULL PRIMARY KEY,
    MANAGER CHAR(10) REFERENCES EMPLOYEE);
```

Similarly, if you added a column FIRSTBOOK to the AUTHORS table, there could be a second relationship between the two tables where the roles are reversed, since a particular BOOKID would then be mentioned within the AUTHORS table for each author as his/her FIRSTBOOK. In that relationship the BOOKS table would be the parent, as shown in Listing 26.

**Listing 26. BOOKS table as parent**

```
ALTER TABLE AUTHORS ADD FIRSTBOOK INTEGER REFERENCES BOOKS;
```

When you define referential constraints on a set of tables, DB2 enforces referential integrity rules on those tables when update operations are performed against them.

- DB2 ensures 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.
DB2 also enforces rules when rows that have dependent rows in a dependent table are deleted from a parent table. The action DB2 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, DB2 does not allow the parent row to be deleted. The rows in dependent tables must be deleted before the row in the parent table.
- 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 null able).
- 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.

**Check constraints**

Check constraints are used to verify that column data conforms rules defined for the column. DB2 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 shown in Listing 27.

**Listing 27. Add column BOOKTYPE with check constraint**

```
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. A `SELECT` statement, when it is run, produces a table as a result. A view essentially is a name for a stored query that you can use much like a table.

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 or store its own data; the data presented in a view is derived from other tables. You can create a view on existing tables, on other views, 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.

The list of views defined is stored in the VIEWS system catalog table. (Note: z/OS the system catalog tables begin with SYSIBM.SYS) The system catalog VIEWDEP table has some number of rows for each view, naming each view or table that view is built upon. Also, each view has an entry in the TABLES system catalog table, and entries in COLUMNS (since views are generally used like tables).

**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, as shown in Listing 28.

**Listing 28. NONFICTIONBOOKS view**

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

After you define this view, there are entries for it in system catalog tables, but there is not any separate storage for the data, as the data within a view is derived from other tables or views.

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 statement shown in Listing 29 creates a MYBOOKVIEW view that contains two columns: TITLE, which represents the BOOKNAME column, and TYPE, which represents the BOOKTYPE column.

**Listing 29. MYBOOKVIEW with two columns**

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

The `DROP VIEW` SQL statement is used to remove a view. Note: For Linux, UNIX, and Windows, if a table or another view on which a view is based is dropped, the dependent view remains defined in the system catalog but becomes inoperative. The VALID column of the VIEWS catalog table indicates whether a view is valid (Y), or not (X). On z/OS, dependent views are dropped when the underlying table or view is dropped.

You can drop the NONFICTIONBOOKS view via:

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

**Read-only and updatable views**

When you create a view, due to its structure it may be either a read-only view or 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 rules for whether a view is updatable are complex and depend on the `SELECT` statement in the definition. For example, views that use `VALUES`, `DISTINCT`, or any form of join are not directly updatable. You can easily determine whether a view is updatable by looking at the `READONLY` column of the `VIEWS` system catalog table: Y means it is read-only and N means it is not.

The detailed rules for creating updatable views are documented in the DB2 SQL Reference (see the Related topics section).

There is a mechanism to allow the appearance of updating data through a read-only view: INSTEAD OF triggers. These triggers can be defined on a view to intercept UPDATE, INSERT, and DELETE against a view, and instead perform actions against other tables, most commonly the base tables the view is built upon.

An updatable view which selects a subset of data from a table may be able to insert data which is not to be included in that subset. For example, 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', DB2 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 DB2 to check that statements using the view result in data that satisfies the conditions of the view, as shown in Listing 30.

### Listing 30. View using WITH CHECK OPTION

```sql
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 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 statements shown in Listing 31, for instance, are no longer allowed.

### Listing 31. Statements not allowed

```sql
INSERT INTO NONFICTIONBOOKS VALUES (...,'F');
UPDATE NONFICTIONBOOKS SET BOOKTYPE = 'F' WHERE BOOKID = 111;
```
Nested views with check option

A view built on top of another view is a nested view. 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, as shown in Listing 32.

Listing 32. View created without check option

```sql
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 shown in Listing 33 would not be allowed because they do not satisfy the conditions of at least one of the views.

Listing 33. INSERT statements not allowed

```sql
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.

```sql
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, as shown in Listing 34.

Listing 34. Satisfying conditions with check option specified

```sql
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 shown in Listing 35 would not be allowed because they do not satisfy the BOOKID > 100 condition of the NONFICTIONBOOKS2 view.
Listing 35. INSERT statements not allowed

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

However, the following INSERT statements shown in Listing 36 would be allowed even though the value N does not satisfy the BOOKTYPE = 'N' condition of the NONFICTIONBOOKS view.

Listing 36. INSERT statements allowed

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

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 DB2 optimizer uses indexes to improve performance when performing queries by minimizing the number of rows to be searched. The optimizer can also eliminate a sort to present the results of a query in the order supported by an index.

Indexes can be defined as unique or non-unique. Non-unique indexes allow duplicate key values, and unique indexes allow only one occurrence of a key value in the list. Unique indexes allow a single null value to be present. However, a second null value would cause a duplicate and therefore is not allowed. Note: z/OS allows UNIQUE WHERE NOT NULL index, which allows many rows with null values.

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 implicitly, the key data is checked for uniqueness and the operation fails if duplicates or any null values are found.

Indexes are created with individual fields as ascending or descending. 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 non-unique 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 specify each column in an index with either ASCENDING or DECENDING. 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, as shown in Listing 37.

**Listing 37. Sorting order**

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

When an index is created, 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 key, for example, can be used to determine the result of the MIN column function.

DB2 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. DB2 reads each row to extract the keys, sort those keys, and then writes the list to the database. If the table is large, then a temporary tablespace is used to sort the keys.

The index is stored in a tablespace. Note: Linux, UNIX, and Windows, separates the indexes into a separate tablespace, then identifies the tablespace when you create the table by 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.

DB2 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 then 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, DB2 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, as shown in Listing 38.

**Listing 38. Clustering index**

```
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, DB2 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. (Note: z/OS permits an expression based upon columns of the table to be included in an index.)

Suppose that you often need to get a list of book names ordered by BOOKID. The query would look like what is shown in Listing 39.

Listing 39. Query for books ordered by BOOKID

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

You can create an index that might improve performance, as shown in Listing 40.

Listing 40. Index to improve performance

```sql
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. DB2 provides a tool to help you estimate the size of an index.
- Indexes are additional copies of the values so they must be maintained as the data in the table is changed via INSERT, UPDATE and DELETE. If table data is frequently changed, consider what impact maintenance of additional indexes may have on performance.
- Indexes can significantly improve performance of particular queries when defined on the appropriate columns.

Note: Optim Query Tuner can provide detailed and specific advice on which indexes to define for either DB2 for Linux, Unix, and Windows, or for DB2 for z/OS.
Triggers

Triggers are snippets of code associated with a table that execute when an INSERT, UPDATE or DELETE action is performed upon that table.

There are also triggers which can be defined upon a view to allow appropriate alternative actions when the view is used for INSERT, UPDATE, DELETE activity, called INSTEAD OF triggers.

Triggers can be defined to perform the actions BEFORE or AFTER each ROW or each STATEMENT.

Several triggers can be defined for the same action on a table. Only one INSTEAD OF can be defined for each action on a view. When several triggers are defined for the same action on a table, they are executed in the same order that they were defined.

Actions that can be taken by a trigger include taking different actions based upon the values involved, affecting the contents of other tables, invoking stored procedures, or returning user-defined error conditions.

For example, given the prior AUTHORS table, with integer columns NONFICTIONBOOKS and FICTIONBOOKS that are summaries of the counts by BOOKTYPE for that author in BOOKS, one could define one of the triggers needed on the BOOKS table to maintain the summaries in the AUTHORS table, as shown in Listing 41.

Listing 41. Creating triggers on the BOOKS table

CREATE TRIGGER BOOKSIA
AFTER INSERT ON BOOKS REFERENCING NEW AS NEW FOR EACH ROW
MODE DB2SQL
WHEN (NEW.BOOKTYPE = 'N') UPDATE AUTHORS
SET NONFICTIONBOOKS=NONFICTIONBOOKS+1 WHERE AUTHORID=NEW.AUTHORID;

To remove the trigger, issue: DROP TRIGGER BOOKSIA;

Conclusion

This tutorial was designed to familiarize you with the features of data types, tables, constraints, views, indexes, and triggers defined in DB2. It also showed you how to use the CREATE, ALTER, and DROP statements to manage these objects. Examples were provided so you could try your hand at using these objects in a controlled setting.

Part 6: Data concurrency, introduces you to the concept of data consistency and to the various mechanisms that are used by DB2 to maintain database consistency in both single-user and multi-user environments.
Related topics

- Read the Preparation Guide for DB2 10.1 Fundamentals Exam 610 to learn in-depth information about each of the concepts presented in this tutorial. This guide is a compilation of topics from the DB2 10.1 Information Center documentation.
- Use an RSS feed to request notification for the upcoming tutorials in this series. (Find out more about RSS feeds of developerWorks content.)
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.
- Evaluate IBM products in the way that suits you best.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)