DB2 9 Fundamentals exam 730 prep, Part 4: Working with DB2 data

Skill Level: Introductory

Roman Melnyk (roman_b_melnyk@hotmail.com)
Staff Information Development
IBM Toronto

24 Jul 2006

This tutorial introduces you to Structured Query Language (SQL) and helps to give you a good understanding of how DB2® 9 uses SQL to manipulate data in a relational database. This tutorial is the fourth in a series of seven tutorials that you can use to help prepare for DB2 9 Fundamentals Certification (Exam 730).

Section 1. Before you start

About this series

Thinking about seeking certification on DB2 fundamentals (Exam 730)? If so, you've landed in the right spot. This series of seven DB2 certification preparation tutorials covers all the basics -- the topics you'll need to understand before you read the first exam question. Even if you're not planning to seek certification right away, this set of tutorials is a great place to begin learning what's new in DB2 9.

About this tutorial

This tutorial explains how DB2 uses SQL to manipulate data in a relational database.. The material provided herein primarily covers the objectives in Section 4 of the exam, which is entitled "Working with DB2 Data using SQL and XQuery." You can view these objectives at: http://www-03.ibm.com/certify/tests/obj730.shtml.

Topics covered in this tutorial include:

- An introduction to SQL

- A description of Data Manipulation Language (DML) and examples that demonstrate how to use it to obtain specific information.
- Transaction boundaries
- SQL procedures and user-defined functions

Objectives

After completing this tutorial, you should be able to:

- Understand the fundamentals of SQL, with a focus on SQL language elements
- Use DML to select, insert, update, or delete data
- Use COMMIT and ROLLBACK statements to manage transactions, and know what constitutes a transaction boundary
- Create and call SQL procedures or invoke user-defined functions from the command line

System Requirements

If you haven't already done so, you can download the free trial version of IBM DB2 9 to work along with this tutorial. Installing DB2 helps you understand many of the concepts that are tested on the DB2 9 Fundamentals Certification exam. DB2 installation is not covered in this tutorial, but the process is documented in the DB2 Information Center.

Conventions used in this tutorial

The following text highlighting conventions are used in this tutorial:

- `Monospaced` text is used for SQL statements. UPPERCASE text identifies SQL keywords, and `lowercase` text identifies user-supplied values in example code.
- Except in the code, database object names are given in uppercase characters, and table column names are given in mixed case.
- *Italic* text is used when introducing a new term, or to identify a parameter variable.

All of the examples in this tutorial are based on the SAMPLE database, which comes with DB2. Because sample output is provided in most cases, you do not need access to the product to understand the examples.

Section 2. Structured Query Language (SQL)

The parts of speech of SQL

SQL is a language that is used to define and manipulate database objects. Use SQL to define a database table, insert data into the table, change the data in the table, and retrieve data from the table. Like all languages, SQL has a defined syntax and a set of language elements.

Most SQL statements contain one or more of the following language elements:

- Single-byte *characters* can be a letter (A-Z, a-z, $, #, and @, or a member of an extended character set), a digit (0-9), or a special character (including the comma, asterisk, plus sign, percent sign, ampersand, and several others).
- A *token* is a sequence of one or more characters. It cannot contain blank characters unless it is a delimited identifier (one or more characters enclosed by double quotation marks) or a string constant.
- An SQL *identifier* is a token that is used to form a name.
- The *data type* of a value determines how DB2 interprets that value. DB2 supports many built-in data types and also supports user-defined types (UDTs).
- A *constant* specifies a value. They are classified as character, graphic, or hexadecimal *string constants,* or integer, decimal, or floating-point *numeric constants.*
- A *special register* is a storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. Examples of special registers are `CURRENT DATE`, `CURRENT DBPARTITIONNUM`, and `CURRENT SCHEMA`.
- A *routine* can be a function, a method, or a procedure.
  - A *function* is a relationship between one or more input data values and one or more result values. Database functions can be either built-in or user-defined.
    Column (or *aggregate*) *functions* operate on a set of values in a column to return a single value. For example:
    - `SUM(sales)` returns the sum of the values in the Sales column.
    - `AVG(sales)` returns the sum of the values in the Sales

column divided by the number of values in that column.

- `MIN(sales)` returns the smallest value in the Sales column.
- `MAX(sales)` returns the largest value in the Sales column.
- `COUNT(sales)` returns the number of non-null values in the Sales column.

Scalar functions operate on a single value to return another single value. For example:

- `ABS(-5)` returns the absolute value of -5 -- that is, 5.
- `HEX(69)` returns the hexadecimal representation of the number 69 -- that is, 45000000.
- `LENGTH('Pierre')` returns the number of bytes in the string "Pierre" -- that is, 6. For a GRAPHIC string, the LENGTH function returns the number of double-byte characters.
- `YEAR('03/14/2002')` extracts the year portion of 03/14/2002 -- that is, 2002.
- `MONTH('03/14/2002')` extracts the month portion of 03/14/2002 -- that is, 3.
- `DAY('03/14/2002')` extracts the day portion of 03/14/2002 -- that is, 14.
- `LCASE('SHAMAN')` or `LOWER('SHAMAN')` returns a string in which all of the characters have been converted to lowercase characters -- that is, 'shaman'.
- `UCASE('shaman')` or `UPPER('shaman')` returns a string in which all of the characters have been converted to uppercase characters -- that is, 'SHAMAN'.

User-defined functions are registered to a database in the system catalog (accessible through the `SYSCAT.ROUTINES` catalog view) using the `CREATE FUNCTION` statement.

- A *method* is also a relationship between a set of input data values and a set of result values. However, database methods are defined, either implicitly or explicitly, as part of the definition of a user-defined structured type. For example, a method called CITY (of type ADDRESS) is passed an input value of type `VARCHAR`, and the result is a subtype of ADDRESS. User-defined methods are registered to a database in the system catalog (accessible through the `SYSCAT.ROUTINES` catalog view) using the `CREATE METHOD` statement. For more information about structured types, refer to *An introduction to structured data types and typed tables* .
- A *procedure* is an application program that can be started by executing a `CALL` statement. The arguments of a procedure are

individual scalar values that can be of different types and that can be used to pass values into the procedure, receive return values from the procedure, or both. User-defined procedures are registered to a database in the system catalog (accessible through the `SYSCAT.ROUTINES` catalog view) using the `CREATE PROCEDURE` statement.

- An *expression* specifies a value. There are string expressions, arithmetic expressions, and case expressions, which can be used to specify a particular result based on the evaluation of one or more conditions.
- A *predicate* specifies a condition that is true, false, or unknown about a given row or group. There are subtypes:
  - A *basic predicate* compares two values (for example, *x* > *y*).
  - The `BETWEEN` predicate compares a value with a range of values.
  - The `EXISTS` predicate tests for the existence of certain rows.
  - The `IN` predicate compares one or more values with a collection of values.
  - The `LIKE` predicate searches for strings that have a certain pattern.
  - The `NULL` predicate tests for null values.

Section 3. Data Manipulation Language (DML)

Using the SELECT statement to retrieve data from database tables

The `SELECT` statement is used to retrieve table or view data. In its simplest form, the `SELECT` statement can be used to retrieve all the data in a table. For example, to retrieve all the STAFF data from the SAMPLE database, issue the following command:

```
SELECT * FROM staff
```

Here is a partial result set returned by this query:

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	-
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	-

To restrict the number of rows in a result set, use the `FETCH FIRST` clause. For example:

```
SELECT * FROM staff FETCH FIRST 10 ROWS ONLY
```

Retrieve specific columns from a table by specifying a *select list* of column names separated by commas. For example:

```
SELECT name, salary FROM staff
```

Use the `DISTINCT` clause to eliminate duplicate rows in a result set. For example:

```
SELECT DISTINCT dept, job FROM staff
```

Use the AS clause to assign a meaningful name to an expression or an item in the select list. For example:

```
SELECT name, salary + comm AS pay FROM staff
```

Without the AS clause, the derived column would have been named 2, indicating that it is the second column in the result set.

Using the WHERE clause and predicates to limit the amount of data returned by a query

Use the `WHERE` clause to select specific rows from a table or view by specifying one or more selection criteria, or search conditions. A *search condition* consists of one or more predicates. A predicate specifies something about a row that is either true or false (see The parts of speech of SQL). When building search conditions, be sure to:

- Apply arithmetic operations only to numeric data types
- Make comparisons only among compatible data types
- Enclose character values within single quotation marks
- Specify character values exactly as they appear in the database

Let's look at some examples.

- Find the names of staff members whose salaries are greater than $20,000:

```
"SELECT name, salary FROM staff
  WHERE salary > 20000"
```

Enclosing the statement within double quotation marks keeps your operating system from misinterpreting special characters, such as * or >; the greater-than symbol could be interpreted as an output redirection request.

- List the name, job title, and salary of staff members who are not managers and whose salary is greater than $20,000:

```
"SELECT name, job, salary FROM staff
  WHERE job <> 'Mgr'
  AND salary > 20000"
```

- Find all names that start with the letter S:

```
SELECT name FROM staff
  WHERE name LIKE 'S%'
```

In this example, the percent sign (%) is a wild card character that represents a string of zero or more characters.

A *subquery* is a `SELECT` statement that appears within the `WHERE` clause of a main query and feeds its result set to that `WHERE` clause. For example:

```
"SELECT lastname FROM employee
  WHERE lastname IN
  (SELECT sales_person FROM sales
    WHERE sales_date < '01/01/1996')"
```

A *correlation name* is defined in the `FROM` clause of a query and can serve as a convenient short name for a table. Correlation names also eliminate ambiguous references to identical column names from different tables. For example:

```
"SELECT e.salary FROM employee e
  WHERE e.salary <
  (SELECT AVG(s.salary) FROM staff s)"
```

Using the ORDER BY clause to sort results

Use the `ORDER BY` clause to sort the result set by values in one or more columns. The column names that are specified in the `ORDER BY` clause do not have to be specified in the select list. For example:

```
"SELECT name, salary FROM staff
  WHERE salary > 20000
  ORDER BY salary"
```

Sort the result set in descending order by specifying `DESC` in the `ORDER BY` clause:

```
ORDER BY salary DESC
```

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:

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:

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.

Using the INSERT statement to add new rows to tables or views

The `INSERT` statement is used to add new rows to a table or a view. Inserting a row into a view also inserts the row into the table on which the view is based.

- Use a `VALUES` clause to specify column data for one or more rows. For example:

```
INSERT INTO staff VALUES (1212,'Cerny',20,'Sales',3,90000.00,30000.00)

INSERT INTO staff VALUES (1213,'Wolfrum',20,'Sales',2,90000.00,10000.00)
```

  Or the equivalent:

```
INSERT INTO staff (id, name, dept, job, years, salary, comm)
  VALUES
  (1212,'Cerny',20,'Sales',3,90000.00,30000.00),
  (1213,'Wolfrum',20,'Sales',2,90000.00,10000.00)
```

- Specify a fullselect to identify data that is to be copied from other tables or views. A *fullselect* is a statement that generates a result table. For example:

```
CREATE TABLE pers LIKE staff
```

```
INSERT INTO pers
  SELECT id, name, dept, job, years, salary, comm
    FROM staff
    WHERE dept = 38
```

Using the UPDATE statement to change data in tables or views

The UPDATE statement is used to change the data in a table or a view. Change the value of one or more columns for each row that satisfies the conditions specified by a WHERE clause. For example:

```
UPDATE staff
  SET dept = 51, salary = 70000
    WHERE id = 750
```

Or the equivalent:

```
UPDATE staff
  SET (dept, salary) = (51, 70000)
    WHERE id = 750
```

If you don't specify a WHERE clause, DB2 updates each row in the table or view!

Using the DELETE statement to get rid of data

The DELETE statement is used to delete entire rows of data from a table. Delete each row that satisfies the conditions specified by a WHERE clause. For example:

```
DELETE FROM staff
  WHERE id IN (1212, 1213)
```

If you don't specify a WHERE clause, DB2 deletes all the rows in the table or view!

Using the MERGE statement to combine conditional update, insert, or delete operations

The MERGE statement updates a target table or updatable view using data from a source table. During a single operation, rows in the target that match the source can be updated or deleted, and rows that do not exist in the target can be inserted.

For example, consider the EMPLOYEE table to be the target table that contains up-to-date information about the employees of a large company. Branch offices handle updates to local employee records by maintaining their own version of the EMPLOYEE table called MY_EMP. The MERGE statement can be used to update the EMPLOYEE table with information that is contained in a MY_EMP table, which is the

source table for the merge operation.

The following statement inserts a row for new employee number 000015 into the MY_EMP table.

```
INSERT INTO my_emp (empno, firstnme, midinit, lastname, workdept,
  phoneno, hiredate, job, edlevel, sex, birthdate, salary)
  VALUES ('000015', 'MARIO', 'M', 'MALFA', 'A00',
  '6669', '05/05/2000', 'ANALYST', 15, 'M', '04/02/1973', 59000.00)
```

And the following statement inserts updated salary data for existing employee number 000010 into the MY_EMP table.

```
INSERT INTO my_emp (empno, firstnme, midinit, lastname, edlevel, salary)
  VALUES ('000010', 'CHRISTINE', 'I', 'HAAS', 18, 66600.00)
```

At this point, the inserted data exists only in the MY_EMP table because it has not yet been merged with the EMPLOYEE table. Following is the MERGE statement that takes the contents of the MY_EMP table and integrates them with the EMPLOYEE table.

```
MERGE INTO employee AS e
  USING (SELECT
   empno, firstnme, midinit, lastname, workdept, phoneno,
   hiredate, job, edlevel, sex, birthdate, salary
   FROM my_emp) AS m
  ON e.empno = m.empno
  WHEN MATCHED THEN
    UPDATE SET (salary) = (m.salary)
  WHEN NOT MATCHED THEN
    INSERT (empno, firstnme, midinit, lastname, workdept, phoneno,
     hiredate, job, edlevel, sex, birthdate, salary)
      VALUES (m.empno, m.firstnme, m.midinit, m.lastname,
       m.workdept, m.phoneno, m.hiredate, m.job, m.edlevel,
       m.sex, m.birthdate, m.salary)
```

Correlation names have been assigned to both the source and the target table to avoid ambiguous table references in the search condition. The statement identifies the columns in the MY_EMP table that are to be considered. The statement also specifies the actions that are to be taken when a row in the MY_EMP table is found to have a match in the EMPLOYEE table, or when a row does not have a match.

The following query executed against the EMPLOYEE table now returns a record for employee 000015:

```
SELECT * FROM employee WHERE empno = '000015'
```

And the following query returns the record for employee 000010 with an updated value for the SALARY column.

```
SELECT * FROM employee WHERE empno = '000010'
```

Using the data-change-table-reference clause to retrieve intermediate result sets in the same unit of work

Suppose you want to give employee 000220 a 7% raise and retrieve her old salary, both in the same unit of work (UOW). You could accomplish this by using a `data-change-table-reference` clause, which is part of the `FROM` clause in an SQL statement.

```
SELECT salary FROM OLD TABLE (
  UPDATE employee SET salary = salary * 1.07
  WHERE empno = '000220'
);

SALARY
-----------
   29840.00

  1 record(s) selected.
```

Columns in the target of the data-change operation (insert, update, or delete) become the columns of the intermediate result table, and can be referenced by name (in this case, Salary) in the select list of the query. The keywords `OLD TABLE` specify that the intermediate result table is to contain values that *precede* the data-change operation. The keywords `NEW TABLE` specify that the intermediate result table is to contain values that *immediately follow* the data-change operation (before referential integrity evaluation and the firing of after triggers has taken place). The keywords `FINAL TABLE` specify that the intermediate result table is to contain values that follow the data-change operation, referential integrity evaluation, and the firing of after triggers.

Suppose you have a CUSTOMERS table that is defined as follows:

```
CREATE TABLE customers (
  cust_id INTEGER GENERATED ALWAYS AS IDENTITY (
   START WITH 10001
  ),
  cust_name VARCHAR(12),
  PRIMARY KEY (cust_id)
);
```

The primary key for this table, Cust_ID, is an automatically generated identity column. You can use a `data-change-table-reference` clause to retrieve the generated identity column value that is being used as a customer number.

```
SELECT * FROM FINAL TABLE (
  INSERT INTO customers (cust_name) VALUES ('Lamarr')
);

CUST_ID     CUST_NAME
----------- ------------
      10001 Lamarr

  1 record(s) selected.
```

Section 4. 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.

For more information about savepoint levels and a detailed example illustrating DB2's savepoint support, see Resources.

Section 5. SQL procedures and user-defined functions

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.db2`, that file could be executed to create the SQL procedure:

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

```
db2 -td@ -vf createSQLproc.db2
```

This `db2` 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))
DYNAMIC RESULT SETS 1
LANGUAGE SQL
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. DB2 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 DB2 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 DB2 Universal Database Version 8. All of the operations that were dependent on a C compiler are now performed by DB2-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 DB2 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:

```
db2 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:

```
db2 "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 `db2 =>` 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"	

Creating and using SQL user-defined functions

Create user-defined functions to extend the set of built-in DB2 functions. For example, create functions that evaluate complex mathematical expressions or manipulate strings, and then reference these functions in SQL statements like you would any existing built-in function.

Suppose that you needed a function that returns the area of a circle when the radius of that circle is specified as an argument to the function. Such a function is not available as a built-in DB2 function, but you could create a *user-defined SQL scalar function* to perform this task and reference the function wherever scalar functions are supported within an SQL statement.

```
CREATE FUNCTION ca (r DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN 3.14159 * (r * r);
```

The `NO EXTERNAL ACTION` clause specifies that the function does not take any

action that changes the state of an object that the database manager does not manage. The DETERMINISTIC keyword specifies that the function always returns the same result for a given argument value. This information is used during query optimization. A convenient way to execute the function is to reference it in a query. In the following example, the query is executed (arbitrarily) against the SYSIBM.SYSDUMMY1 catalog view, which only has one row:

```
db2 SELECT ca(96.8) AS area FROM sysibm.sysdummy1

AREA
------------------------
  +2.94374522816000E+004

  1 record(s) selected.
```

You can also create a *user-defined table function*, which takes zero or more input arguments and returns data as a table. A table function can only be used in the FROM clause of an SQL statement.

Suppose that you needed a function that returns the names and employee numbers of all employees that hold a specific job, with the title of that job specified as an argument to the function. The following is an example of a table function that performs this task:

```
CREATE FUNCTION jobemployees (job VARCHAR(8))
  RETURNS TABLE (
   empno CHAR(6),
   firstname VARCHAR(12),
   lastname VARCHAR(15)
  )
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
   SELECT empno, firstnme, lastname
     FROM employee
     WHERE employee.job = jobemployees.job;
```

The following query references the new table function in the FROM clause and passes the job title 'CLERK' as the argument to the function. A correlation name, introduced by the keyword AS, is required by the syntax:

```
db2 SELECT * FROM TABLE(jobemployees('CLERK')) AS clerk

EMPNO  FIRSTNAME    LASTNAME
------ ------------ ---------------
000120 SEAN         O'CONNELL
000230 JAMES        JEFFERSON
000240 SALVATORE    MARINO
000250 DANIEL       SMITH
000260 SYBIL        JOHNSON
000270 MARIA        PEREZ

  6 record(s) selected.
```

Section 6. Summary

This tutorial was designed to introduce you to Structured Query Language (SQL) and to some of the ways that DB2 9 uses SQL to manipulate data in a relational database. It also covered the fundamentals of SQL, including SQL language elements, Data Manipulation Language (DML), SQL procedures, and user-defined functions. Part 5: Working with DB2 objects, which discusses data types, tables, views, and indexes as defined by DB2, will help you understand how to create and use them.

To keep an eye on this series, bookmark the series page, DB2 9 DBA exam 730 prep tutorials.

Resources

Learn

- Check out the other parts of the DB2 9 Fundamentals exam 730 prep tutorial series.
- Certification exam site. Click the exam number to see more information about Exams 730 and 731.
- DB2 9 overview. Find information about the new data server that includes patented pureXML technology.
- DB2 XML evaluation guide: A step-by-step introduction to the XML storage and query capabilities of DB2 9
- Find additional information about DB2 9 and SQL from the DB2 Information Center.
- An IBM DB2 Universal Database "Stinger" Feature Preview: Enhanced Savepoints offers more information on savepoints.
- What's New for SQL PL in the IBM DB2 Universal Database "Stinger" Release explains the DB2 features that reduce the dependency on a C compiler to create SQL procedures.
- Visit the developerWorks DBA Central zone to read articles and tutorials and connect to other resources to expand your database administration skills.
- Check out the developerWorks DB2 basics series, a group of articles geared toward beginning users.

Get products and technologies

- A trial version of DB2 9 is available for free download.
- 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.

About the author

Roman Melnyk
Roman B. Melnyk , Ph.D., is a senior member of the DB2 Information Development team, specializing in database administration, DB2 utilities, and SQL. During more than eleven years at IBM, Roman has written and edited numerous DB2 books, articles, and other related materials. Roman coauthored *DB2 Version 8: The Official Guide*, *DB2: The Complete Reference*, *DB2 Fundamentals Certification for Dummies*, and *DB2 for Dummies*. Roman recently edited *Apache Derby -- Off to the Races*.