This tutorial shows you how to use SQL statements such as SELECT, INSERT, UPDATE, DELETE, and MERGE to manage data in tables of a SAMPLE database. It also shows how to perform transactions by using the COMMIT, ROLLBACK, and SAVEPOINT statements, how to create stored procedures and user-defined functions, and how to use temporal tables. This is the fourth tutorial in the DB2 10.1 fundamentals certification exam 610 prep series of total six tutorials.

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. The DB2 10.1 fundamentals certification exam 610 prep series of six tutorials cover all the topics you need to know before you take the DB2 10.1 fundamentals certification 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 new features and functionality available in DB2 10 for z/OS® and DB2 10.1 for Linux®, UNIX®, and Windows®.

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
Roughly 20 percent of the DB2 10.1 fundamentals certification exam 610 is designed to test your ability on working with DB2 data using SQL. The content of this tutorial provides basic information
about how to use SQL statements to manage data and perform transactions. It also covers how to create SQL procedures, user-defined functions, and temporal tables. For more in-depth knowledge about working with DB2 data using SQL, see the Related topics section.

Objectives

The content of the tutorial covers the objectives in Section 4 of the DB2 10.1 fundamentals certification exam 610. After completing this tutorial, you should be able to demonstrate knowledge about the following tasks:

- Select, sort, and group data from tables.
- Manage data in tables using SQL statements.
- Commit and rollback transactions.
- Create, call, and retrieve results from DB2 SQL procedures and user-defined functions.
- Identify results from an XQuery.
- Create, manage data, and query temporal tables.

You can also find these objectives listed on IBM certification web site under the Section 4 - Working with DB2 Data using SQL in the exam objectives.

Prerequisites

Understanding the content in this tutorial requires knowledge of the following concepts:

- Basic understanding of relational databases and SQL.
- Installation of DB2 10.1 for Linux, UNIX, and Windows.

System requirements

To run the examples and commands presented in this tutorial, you need access to the SAMPLE database. If you do not have access, you can download a trial copy of DB2 10.1 for personal use and create a SAMPLE database.

After creating the SAMPLE database, you can test the SQL examples in this tutorial. All the examples used in this tutorial have a " (tick) embedded.

Retrieving data from tables

Use the SELECT statement to get data from different types of tables, such as alias tables, temporary tables, hierarchy tables, detached tables, nicknames, materialized query tables, typed tables, and views.

You can issue the SELECT statement from the command line processor (CLP), command line window, or IBM Data Studio after connecting to the sample database.

- To retrieve data from a table in the SAMPLE database, issue the SELECT statement enclosed with double quotes, as shown in this example:

  "SELECT * FROM sales"
Table 1 shows an extract of the results returned by the SELECT statement:

Table 1. Result set from a query that retrieves all rows and all columns of a table

<table>
<thead>
<tr>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>REGION</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/2005</td>
<td>LUCCHESSI</td>
<td>Ontario-South</td>
<td>1</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Ontario-South</td>
<td>3</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Quebec</td>
<td>1</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Manitoba</td>
<td>2</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>GOUNOT</td>
<td>Quebec</td>
<td>1</td>
</tr>
</tbody>
</table>

- To retrieve certain columns of the SALES table, specify the column names separated by comma as follows:
  SELECT sales_date, sales_person, region FROM sales

Table 2 shows an extract of the results returned by the SELECT statement:

Table 2. Result set from a query that retrieves all rows and some specific columns of a table

<table>
<thead>
<tr>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/2005</td>
<td>LUCCHESSI</td>
<td>Ontario-South</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Ontario-South</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Quebec</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>Manitoba</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>GOUNOT</td>
<td>Quebec</td>
</tr>
</tbody>
</table>

Getting a limited amount of data

To retrieve a specific number of rows from a table, use the SELECT statement with the FETCH FIRST clause. This example shows how to retrieve the first two rows from the sales table:

SELECT * FROM sales FETCH FIRST 2 ROWS ONLY

Accessing restricted data

To retrieve uncommitted data from the sales table, use the SELECT statement with the WITH UR clause to indicate UR isolation level as shown in this example:

SELECT * FROM sales WITH UR

To retrieve data from a table with minimal locking, use the FOR FETCH ONLY or FOR READ ONLY clause as shown in this example:
Restricting the result set

To restrict the data in the result set, use one or multiple predicates by specifying the WHERE clause. You can combine predicates by using boolean operators like 'AND'. To get all the employees from the employee table who are hired after year 2005 and whose workdept is in 'AOO' and 'E21', issue a query as shown in this example:

```
SELECT * FROM employee WHERE YEAR(hiredate) > '2005' AND workdept IN ('AOO','E21')
```

Searching for string patterns

To search a specific string in column values, use the LIKE predicate. To find all employees whose first name starts with 'E' in the employee table, issue a query that uses the LIKE predicate with a pattern expression with the % character as shown in this example:

```
SELECT * FROM employee WHERE firstnme LIKE 'E%
```

Eliminating duplicates

To eliminate duplicates from the final result set, use the SELECT DISTINCT clause. To select all the names of salespersons from the sales table without duplicates, issue a query as shown in this example:

```
SELECT DISTINCT sale_person FROM sales
```

Table 3 shows an extract of the result set returned by this query:

<table>
<thead>
<tr>
<th>SALES_PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOUNOT</td>
</tr>
<tr>
<td>LEE</td>
</tr>
<tr>
<td>LUCCHESI</td>
</tr>
</tbody>
</table>

Using functions and expressions

DB2 provides built-in functions that are classified as aggregate functions, scalar functions, or table functions. Use these functions in queries to perform evaluations and return a scalar value or a result set.

You can also specify expressions in the queries. Use expressions to specify a value.

To combine expressions and functions in the same query, issue a query as shown in this example:

```
SELECT SUM(salary+bonus+comm) FROM salary
```
Table 4 shows the result set returned by this query:

### Table 4. Result set from a query that uses expressions and built-in functions

<table>
<thead>
<tr>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2558223.00</td>
</tr>
</tbody>
</table>

#### Specifying data ranges

To restrict the result set by specifying a data range in your query, use the BETWEEN predicate. To select employees whose hire date is between 1998 and 2000, issue a query as shown in this example:

```sql
SELECT firstnme FROM employee WHERE YEAR(hiredate) BETWEEN '1998' and '2000'
```

#### Identifying columns with specific values

To search for columns that have null values, use the NULL predicate. To select all employees without a middle initial, issue a query as shown in this example:

```sql
SELECT firstnme FROM employee WHERE midinit IS NULL
```

To search the employees whose last name is not null, issue a query as shown in this example:

```sql
SELECT firstnme FROM employee WHERE lastname is not null
```

You can also search for expressions that return a negative value. To search for employees whose age is not more than 60 years, issue a query as shown in this example:

```sql
SELECT firstnme FROM employee WHERE YEAR(current date)-YEAR(BIRTHDATE ) < 60
```

#### Retrieving data from multiple tables

To retrieve data from more than one table, use either a cartesian product or join of columns of the same data types.

**Cartesian product**

A cartesian product merges all the values from two tables in one result set. A cartesian product happens when you specify multiple tables in the FROM clause without a WHERE clause. For example, the cartesian product of the following query returns a total row count of 630 because the EMPLOYEE table has 42 rows and the DEPARTMENT table has 15 rows:

```sql
SELECT * FROM employee, department
```

**Joins**

To get data from more than one table, you can use a join predicate on columns of the same data type. If the join columns have indexes, query performance can be improved. You can
provide explicit join operators, such as INNER or LEFT OUTER JOIN, to determine how tables are used in the join.

**Inner joins**
An inner join or equi join returns the rows where the join condition is true in both the tables. This example returns the department number, department name, and first name for each employee where the department number in the employee table is found in the department table. The employees that have a department number in the employee table without a match in the department table are not listed in the result set.

```
SELECT deptno, deptname, firstname FROM department, employee WHERE deptno=workdept
```

You can also use the INNER JOIN operator in the WHERE clause to return the same result set, as shown in this example:

```
SELECT deptno, deptname, firstname FROM department INNER JOIN employee ON deptno=workdept
```

**Table 5** shows an extract of the result set returned by this query:

### Table 5. Result set from a query that uses an inner join

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>FIRSTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>CHRISTINE</td>
</tr>
<tr>
<td>B01</td>
<td>PLANNING</td>
<td>MICHAEL</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>SALLY</td>
</tr>
<tr>
<td>E01</td>
<td>SUPPORT SERVICES</td>
<td>JOHN</td>
</tr>
<tr>
<td>D11</td>
<td>MANUFACTURING SYSTEMS</td>
<td>IRVING</td>
</tr>
<tr>
<td>D21</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>EVA</td>
</tr>
<tr>
<td>E11</td>
<td>OPERATIONS</td>
<td>EILEEN</td>
</tr>
<tr>
<td>E21</td>
<td>SOFTWARE SUPPORT</td>
<td>THEODORE</td>
</tr>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>VINCENZO</td>
</tr>
</tbody>
</table>

**Outer joins**
An outer joins returns rows that match the join condition and the row from both tables that do not match the join condition. These are the three types of outer joins:

- A **Left outer join** returns the matching rows from both tables and the rows in the left table that do not match the join condition, as shown in this example:
  ```
  SELECT deptno, deptname, firstname FROM department LEFT OUTER JOIN employee ON deptno=workdept
  ```

- A **Right outer join** returns the matching rows from both tables and the rows in the right table that do not match the join condition, as shown in this example:
  ```
  SELECT deptno, deptname, firstname FROM department RIGHT OUTER JOIN employee ON deptno=workdept
  ```

- A **Full outer join** returns the matching rows from both tables and the rows in both tables that do not match the join condition, as shown in this example:
  ```
  SELECT deptno, deptname, firstname FROM department FULL OUTER JOIN employee ON deptno=workdept
  ```
In DB2 databases, you can combine more than one outer join in complex queries.

**Using SET operators in queries**

Use the UNION, INTERSECTION, and EXCEPT set operators to combine result sets from SELECT statements:

- **Use** **UNION** **to combine two sets of values and eliminate duplicates**, as shown in this example:

  ```sql
  SELECT sales_person, MAX(sales) FROM sales GROUP BY sales_person
  UNION
  SELECT sales_person, MIN(sales) FROM sales GROUP BY sales_person
  ```

  However, to retrieve all the rows in the result set including duplicates, use UNION ALL.

- **Use** **INTERSECT** **to combine answers from two different sets**. It returns the common values between the two sets. To list all the employees whose resumes are in emp_resume table, use a query as shown in this example:

  ```sql
  SELECT empno FROM emp_resume INTERSECT SELECT empno FROM employee
  ```

  INTERSECT ALL gives all the values.

- **Use** **EXCEPT** **to retrieve the rows that are not present in another result set**. To determine how many employees do not have a project assigned, issue a query, as shown in this example:

  ```sql
  SELECT empno FROM employee EXCEPT SELECT empno FROM empproject
  ```

**Sorting and grouping data in result sets**

The SELECT statement includes clauses to order and group data.

**Sorting data**

Use the ORDER BY clause to sort the data in the result set. If multiple sort keys are specified, they are applied in the order of specification. You can indicate ascending order or descending order. The default is ascending order. To order the result set by first name and then by last name in ascending order, issue a query, as shown in this example:

```sql
SELECT firstname, midinit, lastname FROM employee ORDER BY firstname, lastname
```

You can specify the position of a column in the result set instead of the column name in the ORDER BY clause. To order the result set using the firstname column in ascending order and the lastname column in descending order, issue a query, as shown in this example:

```sql
SELECT firstname, midinit, lastname FROM employee ORDER BY 1 asc, 3 desc
```

**Grouping data**

Use the GROUP BY clause to aggregate values in a SELECT statement. To determine the average sales of a sales person in sales table, issue a query as shown in this example:

```sql
SELECT sales_person, AVG(sales) avg_sales FROM sales GROUP BY sales_person
```

*Table 6* shows an extract of the result set returned by this query:
Table 6. Result set from a query that uses GROUP BY

<table>
<thead>
<tr>
<th>SALES_PERSON</th>
<th>AVG_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOUNOT</td>
<td>3</td>
</tr>
<tr>
<td>LEE</td>
<td>5</td>
</tr>
<tr>
<td>LUCECHESI</td>
<td>1</td>
</tr>
</tbody>
</table>

In this example, the sales_person is grouped into a subset where the average sales are calculated. Specify all columns that are not aggregated in GROUP BY clause. If you use the aggregate functions such as MIN() or MAX() to specify a condition, you must use the HAVING clause. You cannot use the WHERE clause. To determine the minimum and maximum salary paid for a job and that the maximum salary is greater than 27000, issue a query, as shown in this example:

```
SELECT job, MIN(salary), MAX(salary)
FROM employee
GROUP BY job
HAVING MAX(salary) >= 27000
```

Online analytical processing uses a different level of grouping within the same data. DB2 has several OLAP grouping functions, such as GROUPING SETS, CUBE, ROLLUP, and GROUP BY. GROUPING SETS and GROUP BY combine individual rows into a certain fashion as specified.

The GROUPING aggregate function is used in conjunction with super-group functions, such as GROUP BY, CUBE, or ROLLUP. The GROUPING function identifies summary rows in CUBE and ROLLUP query results. It returns 0 or 1 to indicate whether a row returned was generated by the super-group. Returned value of 1 means the row was the result of subtotal and 0 means the row was not the result of subtotal. To calculate how many units are sold by date by sales person, issue a query as shown in this example:

```
"SELECT sales_date, sales_person,
    SUM(SALES) as units_sold,
    GROUPING (sales_date) as date_group,
    GROUPING(sales_person) as sales_group
FROM sales
WHERE YEAR(sales_date) = '2005'
GROUP BY CUBE(sales_date,sales_person)
ORDER BY sales_date,sales_person"
```

Table 7 shows the result set returned by this query:

Table 7. Result set from a query that uses GROUPING

<table>
<thead>
<tr>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
<th>DATE_GROUP</th>
<th>SALES_GROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/2005</td>
<td>GOUNOT</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LEE</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/31/2005</td>
<td>LUCECHESI</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The last row shows the grand total for both DATE_GROUP and SALES_GROUP. A sales person subtotal is indicated when the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0.

### Manipulating data with SQL

You can add, update, or remove data into tables using SQL statements such as INSERT, UPDATE, DELETE, and MERGE. These statements are part of the data manipulation language (DML).

#### Updating column values in a table or view

The UPDATE statement changes the value of one or more columns in a table or view. If you specify a view, the associated table data is updated. You can use the WHERE clause to update a selection of columns or rows that match the condition.

The following example updates the commission to 10 for all the employees with ETHEL as first name:

```
UPDATE employee SET commission = 10 WHERE firstnme = 'ETHEL'
```

If you do not specify a condition with the WHERE clause, all the columns in the table are updated, as shown in this example:

```
UPDATE employee SET (salary, phoneno) = (900000.50, '8888')
```

You can update a table using calculated values from a subquery, as shown in this example:

```
UPDATE employee EMP set ( EMP.salary, EMP.comm) =
(SELECT avg(salary), avg(comm) FROM employee WHERE firstnme = 'ETHEL')
```

#### Deleting data from a table or view

Use the DELETE statement to eliminate data from a table or view. Deleting rows from a view removes the corresponding data from the base table.

To remove specific rows that match a condition, use the WHERE clause. To delete employees whose name starts with 'J', issue the DELETE statement with the WHERE clause, as shown in this example:

```
DELETE FROM employee WHERE firstnme LIKE 'J%'
```

You can use a subquery in the condition to specify which rows to delete, as shown in this example:
DELETE FROM employee WHERE lastname IN (SELECT sales_person FROM sales WHERE YEAR(sales_date) = 1971)

You can delete all rows with one DELETE statement. To remove all the rows from the employee table, issue the DELETE statement without a WHERE clause, as shown in this example:

DELETE FROM employee

For a faster delete operation, use the TRUNCATE TABLE statement to delete all the data:

TRUNCATE TABLE employee IMMEDIATE

Adding data to tables, nicknames, or views
Use the INSERT statement to add a new row into a table, nickname, or view. Inserting a row into nickname or view inserts the row into the base table.

To insert a row into a table that has columns with the NOT NULL constraint, you must specify values for all these columns. To insert data to the act table, all the columns with the NOT NULL constraint are specified in the INSERT statement, as shown in this example:

```
INSERT INTO act VALUES (190,'DBA','CREATE DATA')
```

You can specify column names in the INSERT, as shown in this example for the table department:

```
"INSERT INTO department (deptno, deptname, admrdept) VALUES

('B11', 'PURCHASING', 'B01'),
('E41', 'DATABASE ADMINISTRATOR', 'E01')"
```

Updating tables or views with the MERGE statement
Use the MERGE statement to insert, update, or delete rows in a table or view in one atomic block by specifying a condition that determines when to update or delete an existing row and when to add a row that does not exist. Updating, deleting, or inserting a row in a view updates, deletes, or inserts the row in the base tables.

To update the employee table for specific departments to increase the salary by $300.00, issue a MERGE statement, as shown in this example:

```
MERGE INTO employee emp
USING (SELECT deptno FROM department) dep
ON (emp.workdept = dep.deptno )
WHEN MATCHED THEN
UPDATE SET
SALARY = SALARY + 300
WHEN NOT MATCHED THEN
SIGNAL SQLSTATE '70002'
SET MESSAGE_TEXT =
'There are no employees found for department: ' CONCAT dep.deptno
```
The MERGE statement uses an algorithm to improve the performance of the INSERT, UPDATE, and DELETE statements.

**Working with transactions**

For DB2 databases, a transaction is a unit of work (UOW) consisting of a series of sequential SQL statements such as CREATE, INSERT, UPDATE, OR DELETE that ends when a COMMIT or ROLLBACK happens.

**Committing or rolling back UOWs**

When developing applications, you can explicitly set transactions by issuing a COMMIT or ROLLBACK statement after issuing a series of DML statements. When you issue a COMMIT statement, the database manager makes all the changes, within that UOW, to the database permanent. However, if there is an error in a statement within the UOW or you issue a ROLLBACK statement, the database manager reverts all the changes made to database and the database gets to the same state it was before the UOW started.

The autocommit mode is set to true by default so that after every statement, an implicit commit is issued. Every statement is a single complete UOW. When working with large UOWs, it is good practice to issue a commit after a number of statements to prevent loss of work. However, too many commits in a UOW makes it difficult to get back to a consistent data state because it requires crossing lots of transaction boundaries.

**Using savepoints UOWs**

Use the SAVEPOINT statement to define levels within a UOW. By setting a savepoint within a UOW, you can roll back to that savepoint. In the following example, a row for an employee with the name Marino is inserted, the commission for Edwards is set to 190, the row with id 170 is deleted, and the row id 370 is inserted in the staff table:

```sql
INSERT INTO staff (id, name, dept, job, years, salary, comm) VALUES
(360, 'Marino', 84, 'Clerk', 16, 65000, 716)
UPDATE staff SET comm = 190.00 WHERE name = 'Edwards'
SAVEPOINT del
DELETE FROM staff WHERE id = 170
SAVEPOINT ins
INSERT INTO staff (id, name, dept, job, years, salary, comm) VALUES
(370, 'Akshaya', 66, 'Sales', 66, 68900, 1250)
```

Two savepoints are set within the UOW to mark where a DELETE statement is issued and where an INSERT statement is issued. To undo the insert of row id 370 but still delete row id 170 in the staff table, issue a rollback to the `ins` savepoint, as shown in this example:

```sql
ROLLBACK TO ins
```

To undo the insert of row id 370 and undo the delete of row id 170 in the staff table, issue a rollback to the `del` savepoint, as shown in this example:

```sql
ROLLBACK TO del
```
Working with SQL procedures or user-defined functions

You can create SQL procedures in the db2 database server as a way to convert business or system logic to a process that uses SQL and runs on the db2 database server. Use SQL procedures to improve database performance or database security.

DB2 user-defined functions (UDFs) provide a way to create SQL scalar, table, or row functions in the db2 database server. Use UDFs to extend the capabilities of the DB2 built-in functions.

Creating SQL procedures

Use the CREATE PROCEDURE statement to create a new SQL procedure or recreate an existing SQL procedure by using the REPLACE keyword. You must specify a name, attributes, and the procedure body. In addition to certain SQL statements, you can use variables, cursors, exceptions handling, declared global temporary tables, and other capabilities. For a complete list, see SQL statements that can be executed in routines and triggers.

To create an SQL procedure, you must indicate a name that can be up to 128 characters in length. If you do not specify a schema name, the current schema indicated by the CURRENT SCHEMA special register value is used. You can indicate input (IN), output (OUT), or both input and output (INOUT) parameters enclosed in parenthesis and separated by comma. The data types for parameters are the same as the SQL data type and size. However, you can create SQL procedures without parameters, as shown in this example:

```
CREATE PROCEDURE updates_employees BEGIN ... END
```

The procedure name with the combination of parameters has to be unique. A specific name can be used to make a procedure name unique if the procedure has the same name but different parameters. If you do not indicate a specific name, one is automatically generated beginning with the SQL prefix followed by a timestamp string, such as SQL130927120344000.

For complete details about the syntax of the CREATE PROCEDURE statement to declare SQL procedures, see CREATE PROCEDURE (SQL) statement in the DB2 Information Center center.

To create a simple SQL procedure to report the name of staff members, their departments, and their salaries for employees who received more than a given commission, write the SQL statement into a file `check_comm.clp`, as shown in this example:

```
Listing 1. Creating the check_comm procedure

CREATE OR REPLACE PROCEDURE check_comm (IN v_comm SMALLINT, OUT v_sqlstatus CHAR(5))
SPECIFIC check_comm
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE chn_cur CURSOR WITH RETURN FOR SELECT name, dept, job, salary
  FROM staff
  WHERE comm > v_comm;
  OPEN chn_cur;
  SET v_sqlstatus = SQLSTATE;
END@
```

The SPECIFIC clause provides a unique name to identify the SQL procedure. The DYNAMIC RESULT SETS clause indicates 1 as the maximum number of result sets. The LANGUAGE SQL clause indicates that this is an SQL stored procedure. The SQL procedure body is enclosed by the BEGIN and END keywords. It contains a declaration of a variable and a cursor, setting of the output parameter, and returning a result set by opening the cursor.

The v_comm input parameter indicates the commission value. The v_sqlstatus output parameter returns the status for the last SQL statement. If the execution of the SQL procedure is successful, the value of v_sqlstatus is 0. If the execution of the SQL procedure is unsuccessful, the value of v_sqlstatus is the standardized error code.

To create the SQL procedure, run the check_comm.clp file, as shown in this example:

```
db2 -td@ -vf check_comm.clp
```

The -td parameter specifies the @ character as the end of statement character. Avoid using the ; character because it is used as the end of statements in the procedure body. The –v parameter displays the current command in the standard output and the –f parameters indicates the input command file.

Use the SYSCAT.ROUTINES catalog view to show a list of the user-defined routines existing in the database. To show a list of the procedures created by the authorization ID ARADA, issue the following query:

```
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES
WHERE OWNER='ARADA' AND ROUTINETYPE='P'
```

**SQL Procedural Language**

The SQL Procedural Language (SQL PL) provides statements for managing variables and cursors, conditional statements, looping statements, transfer of control statements, condition handler statements, and error management statements.

**Managing variables and cursor statements**

Use the following statements to declare variables and assign values to them:

- The DECLARE <variable> statement in SQL procedures to define variables.
- The SET (assignment-statement) in SQL procedures to assign a value to declared variables. Alternatively, use the SELECT INTO statement for the same purpose.

In SQL procedures, use the cursor statements to define a result set and apply logic to each row in the result set:

- The DECLARE CURSOR statement declares the cursor to define the result set.
- The OPEN statement to initialize the cursor so that it can be used to retrieve data.
- The FETCH statement retrieves data into local variables one row at a time until the end of the result set.
Conditional statements
Use conditional statements to evaluate one or multiple conditions to determine what logic to execute:
- Use the **IF** statement to evaluate a single condition. The **ELSE** keyword specifies what statement to run when the condition is not true. The **ELSEIF** keyword specifies nested IF conditions. You must end every **IF** statement with the **ENDIF** keyword.
- Use the **CASE** statement to evaluate multiple conditions. You must end every **CASE** statement with the **END CASE** keyword.

Looping statements
Use looping statements to iterate the execution of a statement or a group of statements:
- The **FOR** statement runs a statement or group of statements for every row in a result set. You must end every **FOR** statement with the **END FOR** keyword.
- The **WHILE** statement runs a statement or group of statements while the evaluation of the specified condition is **TRUE**. When the condition becomes false, it exits the loop. Every **WHILE** is finished with **END WHILE**.
- The **REPEAT** statement runs a statement or group of statements until the specified search condition becomes true. The **UNTIL** keyword indicates the search condition. You must end every **REPEAT** statement with the **END REPEAT** keyword.
- The **LOOP** statement runs a statement or group of statements. You must specify a terminating condition by using the **GOTO**, **ITERATE**, or **LEAVE** statement in a conditional statement. Otherwise, the **LOOP** statement runs infinitely. You must end every **LOOP** statement with the **END LOOP** keyword.

Transfer of control statements
Use transfer of control statements to redirect the flow of control within an SQL procedure. This unconditional branching is generally used in looping statements. However, they can be used anywhere in the SQL procedure.
- The **LEAVE** statement transfers the control out of a looping or compound SQL statement.
- The **ITERATE** statement brings the control back to beginning of a labeled looping statement.
- The **GOTO** statement transfers the control permanently to a specific label within the stored procedure. This statement should be used cautiously.
- The **RETURN** statement immediately returns the control back to the caller of the SQL procedure.

Conditions, condition handlers, and error management
Conditions declare a named condition with an optional associated SQLSTATE value. Condition handlers determine the behavior of your SQL procedure when a condition occurs. You can declare one or more condition handlers in your SQL procedure for general conditions, named conditions, or specific SQLSTATE values:
- The **DECLARE <condition>** statement declares a named condition and optionally an associated SQLSTATE.
- The **DECLARE <condition handler>** statement defines the actions for a general or named condition.
When a general condition is raised by an SQL statement in an SQL procedure such as SQLEXCEPTION, SQLWARNING, and NOT FOUND, the database manager passes the control to any condition handlers declared in the SQL procedure. You can use different condition handlers such as EXIT, CONTINUE, and UNDO, to catch and manage errors and complete the processing of the SQL procedure.

It's a good practice to check the SQLSTATE after an SQL statement in an SQL procedure. For example, you should check the SQLSTATE after any DECLARE, OPEN, FETCH and CLOSE statement for a cursor. In addition, you should declare a condition handler at the beginning of the SQL procedure to handle any errors that are encountered while the SQL procedure is running.

These statements can help you in error management:

- The **SIGNAL** statement returns and error or warning with the specified SQLSTATE, along with optional message text.
- The **GET DIAGNOSTICS** statement returns information about the current execution environment including information about the previous SQL statement. Issue this statement after the CALL statement to determine the return status of an SQL procedure.

For more details about SQL PL statements and examples, see [Control of flow statement in SQL procedures](https://www.ibm.com/support/knowledgecenter/en/SSLTBW_AIX_81/com.ibm.db2.luw.sql.rer.sql00000.html) in the DB2 Information Center.

**Calling SQL procedures**

Use the CALL statement to run SQL procedures. The name of the procedure and the specified parameters should identify a unique procedure.

To run the SQL procedure `check_comm` that was created in [Listing 1](#), issue the CALL statement from the DB2 CLP or an sql editor in IBM Data Studio, as shown in this example, where 1000 is the input parameter and `?` is the place holder for the `v_sqlstatus` output parameter that returns the latest SQLSTATE value:

```
"CALL check_comm(1000,?)"
```

The following text is returned after the CALL statement is completed. It shows the name and value of the output parameter and the result set from the oper cursor with a list of staff members that have a commission greater than $1000.00.

<table>
<thead>
<tr>
<th>Value of output parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Name : V_SQLSTATUS</td>
</tr>
<tr>
<td>Parameter Value : 00000</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Rothman</td>
</tr>
<tr>
<td>Koonitz</td>
</tr>
<tr>
<td>Edwards</td>
</tr>
</tbody>
</table>
Creating user-defined functions

Use user-defined functions (UDFs) when you want to extend the capabilities of existing DB2 built-in functions.

UDFs provide a way to generate consistent output for given input. It also helps to have some of the logic perform in the data server and thus improve application performance.

You can define the following type of UDFs:

- **Scalar functions** to return a single value.
- **Aggregate functions** to return a single value for a set like values.
- **Row functions** to return one row.
- **Table functions** to return a table (result set).

The DB2 built-in functions include scalar functions such as MIN and MAX, aggregate functions such as AVG and SUM, and table functions such as XMLTABLE. For a complete list, see [Built-in functions](https://www.ibm.com/support/docviewynchronously/docdisplay?ersm Lovcode=No& perceivemodal=Normal&rs=19222) in the DB2 Information Center.

Use the SYSCAT.ROUTINES catalog view to show a list of the user-defined routines existing in the database. To show a list of the user-defined functions created by the authorization ID ARADA, issue the following query:

```sql
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES
WHERE OWNER='ARADA' AND ROUTINETYPE ='F'
```

The UCASE and LCASE built-in functions are of scalar type. They take a string of characters as input returns this string in uppercase or lowercase. To show the first name of an employee in uppercase and lowercase, issue a query on the employee table that uses these scalar functions, as shown in this example:

```sql
SELECT firstname, LCASE(firstname) FROM EMPLOYEE
```

To create a scalar function that calculates the tangent for a given argument, issue the CREATE FUNCTION statement, as shown in this example:

```sql
Listing 2. Creating an scalar UDF function
CREATE OR REPLACE FUNCTION TAN (X DOUBLE)
RETURNS DOUBLE
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN SIN(X)/COS(X)
```

The NO EXTERNAL ACTION clause specifies that the function does not perform an action that changes the state of an object that the database manager does not manage. The DETERMINISTIC clause indicates to the database manager to always return the same results for given argument values.
You can use a table function to return EMPNO, LASTNAME, FIRSTNME, and DEPARTMENT_NAME for all the employees of a given department number. To create the table function, issue the CREATE FUNCTION statement, as shown in Listing 3:

**Listing 3. Creating a UDF table function**

```sql
CREATE OR REPLACE FUNCTION employees_same_dept_name (deptno CHAR(3))
RETURNS TABLE (empno CHAR(6),
lastname  VARCHAR(15),
firstname VARCHAR(12),
department_name varchar(30))
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
RETURN
SELECT empno, lastname, firstnme, deptname
FROM employee, department
WHERE employee.workdept = employees_same_dept_name.deptno
AND employee.workdept = department.deptno
```

**Calling user-defined functions**

To call the function TAN created in **Listing 2**, issue a query to the sysibm.sysdummy1 table, as shown in this example:

```sql
SELECT TAN(10) FROM sysibm.sysdummy1
```

To call the function created in **Listing 3**, issue a query calling this table function with 'A00' as the input parameter for department number, as shown in this example:

```sql
SELECT * FROM TABLE(employees_same_dept_name('A00')) AS emp_same_dpt
```

**Table 8** shows the results of issuing this query:

**Table 8. Result set from a query that uses a table UDF**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>HAAS</td>
<td>CHRISTINE</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>110</td>
<td>LUCCHESI</td>
<td>VINCENZO</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>120</td>
<td>O'CONNELL</td>
<td>SEAN</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>200010</td>
<td>HEMMINGER</td>
<td>DIAN</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>200120</td>
<td>ORLANDO</td>
<td>GREG</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
</tbody>
</table>

**Retrieving data using XQuery**

You can use DB2 databases to store well-formed XML data in a column of a table and use SQL, XQuery, or a combination of SQL and XQuery to retrieve the XML data from the database.
You can retrieve result sets from queries by specifying XQuery expressions in one of the following ways:

- Using XQuery as the primary language.
- Using SQL with the XMLQUERY SQL function.

If you use XQuery directly, each item of a sequence is returned as a separate row of the result set. However, when you use the XMLQUERY SQL function, each resulting sequence from a row in the table is returned as a row in the result set.

For more details and examples, see Data retrieval with XQuery in the DB2 Information Center.

**Working with temporal tables**

Use temporal tables when time is critical as it is in industries such as insurance, travel, and retail. Temporal tables can have data dependent on time in the past, present, or future. The data that is not in temporal tables is associated with present time. Before temporal tables, you could store the time based data using an application which was time-consuming and expensive to develop. With addition of time temporal tables to DB2 10.1, you can simply use SQL queries to obtain time-aware data.

Time temporal tables use two basic concepts:

- System time tracks when changes are made to the tables.
- Business time tracks for which time the row is valid.

To track temporal data for both system time and business time, use bitemporal tables.

Use the SYSCAT.TABLES catalog view to determine whether a table is temporal or not. The following query shows a list of tables with their temporal type:

```sql
SELECT tabschema, tabname, temporatype FROM syscat.tables
```

The following are valid values for the `temporatype` column:

- A – Application period temporal table
- B - Bitemporal table
- N - Not a temporal table
- S - System-period temporal table

Starting with DB2 10.1, the SYSCAT.PERIODS catalog view returns all the definitions of a period for use with a temporal table.

**System-period temporal tables**

System-period temporal tables use a historical version of data. Use a system-period temporal table to store the current version of data. The associated history table automatically stores the updated and deleted data. This type of temporal table must include a SYSTEM_TIME period column to capture the begin and end times when the data in a row is current. It must include a transaction start-ID column to capture the time when execution started for a transaction that impacts the row.
Creating system-period temporal tables
To create a system-period temporal table, use the CREATE TABLE statement with the required columns for a SYSTEM_TIME period and a transaction_id.
To create a product table for a retail store as a system-period temporal table, issue the CREATE TABLE statement, as shown in this example:

```
CREATE TABLE product_info
(
  sku_no      VARCHAR(15) NOT NULL,
  store_id    VARCHAR(19) NOT NULL,
  amt         INT NOT NULL,
  sys_start   TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,
  sys_end     TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,
  ts_id       TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS TRANSACTION START ID,
  PERIOD SYSTEM_TIME (sys_start, sys_end)
) IN PRD_DMS_SPACE;
```

The `sku_no` column stores the product unique number for a particular store and `amt` is its cost. The `sys_start` and `sys_end` columns indicate when the product information in each row becomes active or inactive. The `ts_id` column indicates the start time of a transaction that affects the row.

To create the associated history table, create a table that has the same columns as the system-period temporal table. However, the columns to indicate the period and the transaction_id should not have the GENERATED ALWAYS clause in the create statement. To create a history table for the `product_info` table, issue the following CREATE TABLE statement:

```
CREATE TABLE hist_product_info
(
  sku_no       VARCHAR(15) NOT NULL,
  store_id     VARCHAR(19) NOT NULL,
  amt          INT NOT NULL,
  sys_start    TIMESTAMP(12) NOT NULL ,
  sys_end      TIMESTAMP(12) NOT NULL ,
  ts_id        TIMESTAMP(12) NOT NULL
) IN PRD_DMS_SPACE;
```

You can also create the `hist_product_info` table by using the CREATE TABLE statement with the LIKE clause, as shown in this example:

```
CREATE hist_product_info LIKE product_info;
ALTER TABLE product_info ADD VERSIONING USE HISTORY TABLE hist_product_info
```

After you create the history table, create the link to the system-period temporal table by using the ALTER TABLE statement with ADD VERSIONING clause, as shown in this example:

```
ALTER TABLE product_info ADD VERSIONING USE HISTORY TABLE hist_product_info
```

Inserting data into system-period temporal tables
Use the INSERT statement to add data into system-period temporal tables. Data is added to the `hist_product_info` table when you delete or update data into system-period temporal tables.
To add data for four products into the `product_info` table, issue INSERT statements, as shown in the following example:

```
INSERT INTO product_info (sku_no, store_id, amt) VALUES ('CHR_00001', 'NJ001', 5.99);
INSERT INTO product_info (sku_no, store_id, amt) VALUES ('CHR_00002', 'NJ011', 7.99);
INSERT INTO product_info (sku_no, store_id, amt) VALUES ('CHR_00002', 'NY019', 9.99);
INSERT INTO product_info (sku_no, store_id, amt) VALUES ('CHR_00005', 'NY019', 8.99);
```

Table 9 shows the data added to the `product_info` table:

**Table 9. Inserted data into a system-period table**

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>5.99</td>
<td>09/29/2013</td>
<td>12/30/9999</td>
<td>09/29/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NY019</td>
<td>8.99</td>
<td>09/29/2013</td>
<td>12/30/9999</td>
<td>09/29/2013</td>
</tr>
</tbody>
</table>

Updating data into system-period temporal tables

Use the UPDATE statement to modify data in a system-period temporal table. Automatically, data is added to the associated history table.

To increase the price of the product with sku_no CHR_00001 from 5.99 to 6.99, issue the UPDATE statement in the `product_info` table, as shown in this example:

```
"UPDATE product_info SET amt = 6.99 WHERE sku_no = 'CHR_00001'"
```

Table 10 shows the content of the `product_info` table after this update:

**Table 10. Updated data in a system-period table**

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>6.99</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NY019</td>
<td>8.99</td>
<td>09/29/2013</td>
<td>12/30/9999</td>
<td>09/29/2013</td>
</tr>
</tbody>
</table>

This update in the `product_info` table inserts a row in the `hist_product_info` table to indicate the time when the inserted values were active. Table 11 shows the row added to the `hist_product_info` table after the update operation:

**Table 11. Inserted data into the associated history table**

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>5.99</td>
<td>09/29/2013</td>
<td>09/30/2013</td>
<td>09/29/2013</td>
</tr>
</tbody>
</table>

The `sys_start` column in the `hist_product_info` table has the same value as the `sys_start` column in the `product_info` after the insert (see Table 9). The `sys_end` column of the
hist_product_info table has the same value as the sys_start column in the updated row of the product_info table (see Table 10).

Deleting data from system-period temporal tables

Use the DELETE statement to remove rows from system-period temporal tables. Also, it adds rows to the associated history table with the appropriate system timestamps.

To delete the product with sky_no 'CHR_00005' because it is no longer manufactured, remove the row from the product_info table, as shown in this example:

```sql
DELETE FROM product_info WHERE sku_no = 'CHR_00005'
```

This statement inserts a row in the hist_product_info table where the sys_start column has the same value as the sys_start column in the product_info table and the sys_end column has the time of the delete. Table 12 shows the row added to the hist_product_info table after the delete operation:

Table 12. Inserted data into the associated history table after delete operation

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>5.99</td>
<td>09/29/2013</td>
<td>09/30/2013</td>
<td>09/29/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NY019</td>
<td>8.99</td>
<td>09/29/2013</td>
<td>09/30/2013</td>
<td>09/29/2013</td>
</tr>
</tbody>
</table>

Querying system-period temporal tables

Use the SELECT statement to query a system-period temporal table to retrieve results for a specified point or period in time. Use the following clauses to specify a point in time or time period:

- Use FOR SYSTEM_TIME AS OF value1 to include all the rows with a time period that begins before or on value1 and ends after value1 from both the system-period temporal and history tables, as shown in this example:

```sql
SELECT sku_no, store_id, amt FROM product_info FOR SYSTEM_TIME AS OF '2013-09-29-10.23.50.623709000000'
```

- Use FOR SYSTEM_TIME FROM value1 TO value2 to include all the rows with a time period that begins before or on value1 and ends before value2 from both the system-period temporal and history tables, as shown in this example:

```sql
SELECT sku_no, store_id, amt FROM product_info FOR SYSTEM_TIME FROM '0001-01-01-00.00.00.000000' TO '9999-12-30-00.00.00.000000000000' WHERE sku_no = 'CHR_00001'
```

- Use FOR SYSTEM_TIME BETWEEN value1 AND value2 to include all the rows with a time period that overlaps any point in time between value1 and value2 from both the system-period temporal and history tables, as shown in this example:

```sql
SELECT * FROM product_info FOR SYSTEM_TIME BETWEEN '2013-09-29-10.23.50.623709000000' and '9999-12-30-00.00.00.000000000000' WHERE sku_no = 'CHR_00001'
```

If you issue a query without specifying a time period, it returns only rows in the system-period temporal table, as shown in this example:
SELECT sku_no, store_id, amt FROM product_info WHERE sku_no = 'CHR_00001'

For more information about using system-period temporal tables and examples, see System-period temporal tables in the DB2 Information Center.

**Application-period temporal tables**

Use an application-period temporal table to manage data based on time criteria that defines the time period when data is valid. This type of temporal table must include a BUSINESS_TIME period column to capture the begin and end times when the data in a row is valid. However, there is no history table associated.

**Creating application-period temporal tables**

To create an application-period temporal table, use the CREATE TABLE statement with the required column for a BUSINESS_TIME period. To create a product table for a retail store as an application-period temporal table, issue the CREATE TABLE statement, as shown in this example:

```sql
CREATE TABLE product_info
( sku_no                VARCHAR(15) NOT NULL,
  store_id              VARCHAR(19) NOT NULL,
  amt                   DECIMAL(10,2) NOT NULL,
  bus_start             DATE NOT NULL,
  bus_end               DATE NOT NULL,
  PERIOD BUSINESS_TIME (bus_start, bus_end)
) IN PRD_DMS_SPACE;
```

To prevent overlapping business_time periods, create a unique index and specify the BUSINESS_TIME WITHOUT OVERLAPS clause, as shown in this example:

```sql
CREATE UNIQUE INDEX ix_product_info
    ON product_info (sku_no, store_id, BUSINESS_TIME WITHOUT OVERLAPS)
```

You can change a regular table into an application-period temporal table by using an ALTER TABLE statement to add the time columns and the BUSINESS_TIME period column. If the `product_info` table already existed, issue the ALTER TABLE statement to change it, as shown in this example:

```sql
ALTER TABLE product_info ADD COLUMN bus_start DATE NOT NULL;
ALTER TABLE product_info ADD COLUMN bus_end DATE NOT NULL;
ALTER TABLE product_info ADD PERIOD BUSINESS_TIME(bus_start, bus_end);
ALTER TABLE product_info ADD CONSTRAINT u-index
    UNIQUE(sku_no, store_id, BUSINESS_TIME WITHOUT OVERLAPS);
```

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

Use the INSERT statement to add data into application-period temporal tables. To add data for four periods into the `product_info` table, issue INSERT statements, as shown in the following example:

```sql
INSERT INTO product_info VALUES('CHR_00001', 'NJ01', 11.99, '2013-06-01', '2013-08-01');
```
The third INSERT statement for CHR_00001 fails because it indicates a period that overlaps with the specified period in the second INSERT. You can issue the SELECT * FROM product_info statement to verify the results of the insert operation. Table 13 shows the data added to the product_info table:

**Table 13. Inserted data into an application-period table**

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>9.99</td>
<td>1/1/2013</td>
<td>7/1/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>10.99</td>
<td>7/1/2013</td>
<td>1/1/2014</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NJ01</td>
<td>8.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
</tr>
<tr>
<td>CHR_00007</td>
<td>NJ01</td>
<td>25.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
</tr>
</tbody>
</table>

The results show just two rows for CHR_00001 with different business times because the third INSERT failed.

**Updating application-period temporal tables**

Use the UPDATE statement to modify data in an application-period temporal table. You can update data for a specific time period. This kind of update can cause the splitting of rows and inserting of new rows into the table. Table 13 lists two rows in the product_info table for CHR_00001 with the price for 2013. To change the price of CHR_00001 from June 1st 2013 to Aug 1st 2013, issue the UPDATE statement, as shown in Listing 4:

**Listing 4. Updating data for a specific period of time**

```sql
UPDATE product_info FOR PORTION OF BUSINESS_TIME FROM '2013-06-01' TO '2013-08-01'
SET amt = 14.99 WHERE sku_no = 'CHR_00001'
```

As a result of this update, the two existing rows for CHR_00001 are split. These two rows are updated with the new time period and two new rows are inserted for CHR_00001 with the price 14.99. Table 14 shows the updated data:

**Table 14. Updated data for a specific period of time**

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>9.99</td>
<td>1/1/2013</td>
<td>6/1/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>10.99</td>
<td>8/1/2013</td>
<td>1/1/2014</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>14.99</td>
<td>6/1/2013</td>
<td>7/1/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>14.99</td>
<td>7/1/2013</td>
<td>8/1/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NJ01</td>
<td>8.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
</tr>
<tr>
<td>CHR_00007</td>
<td>NJ01</td>
<td>25.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
</tr>
</tbody>
</table>

**Deleting data from application-period temporal tables**

Use the DELETE statement to remove data from application-period temporal tables. You can delete data for a specific time period as shown in the following example:
DELETE FROM product_info FOR PORTION OF BUSINESS_TIME FROM '2013-06-15' TO '2013-08-15' WHERE sku_no = 'CHR_00001'

This DELETE statement removes the row with bus_start as 07/01/2013 and bus_end as 08/01/2013 and updates existing rows to adjust the time period for sku_no 'CHR_00001'. Table 15 shows the results of this statement:

### Table 15. Updated data after the delete operation

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>9.99</td>
<td>1/1/2013</td>
<td>6/1/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>14.99</td>
<td>6/1/2013</td>
<td>6/15/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NJ01</td>
<td>10.99</td>
<td>8/15/2013</td>
<td>1/1/2014</td>
</tr>
</tbody>
</table>

Querying the application-period temporal tables

Query an application-period temporal table to retrieve results for a specified time period. Use the following clauses to specify a time period:

- Use `FOR BUSINESS_TIME AS OF value1` to include all the rows with a time period that begins before or on `value1` and ends after `value1`, as shown in this example:
  
  ```sql
  SELECT sku_no, amt, bus_start, bus_end FROM product_info
  FOR BUSINESS_TIME AS OF '2013-08-15' WHERE sku_no = 'CHR_00001'
  ```

- Use `FOR BUSINESS_TIME FROM value1 TO value2` to include all the rows with a time period that begins before or on `value1` and ends before `value2`, as shown in this example:
  
  ```sql
  SELECT sku_no, amt, bus_start, bus_end FROM product_info
  FOR BUSINESS_TIME FROM '2013-01-01' TO '2013-06-15' WHERE sku_no = 'CHR_00001'
  ```

- Use `FOR BUSINESS_TIME BETWEEN value1 AND value2` to include all the rows with a time period that overlaps any point in time between `value1` and `value2`, as shown in this example:
  
  ```sql
  SELECT sku_no, amt, bus_start, bus_end FROM product_info
  FOR BUSINESS_TIME BETWEEN '2013-01-01' AND '2013-01-01'
  ```

If you issue a query without specifying a time period, it returns all the rows in the table, as shown in this example:

```sql
SELECT sku_no, amt, bus_start, bus_end FROM product_info WHERE sku_no = 'CHR_00001'
```

For more information about using application-period temporal tables and examples, see Application-period temporal tables in the DB2 Information Center.

**Bitemporal tables**

Use bitemporal tables to keep user-based period information as well as system-based historical information. These tables combine the capability of tracking history of system-period temporal tables and the time-specific storage of application-period tables.
Creating bitemporal tables
To create a bitemporal table, use the CREATE TABLE statement with the required columns for a transaction_id, SYSTEM_TIME, and BUSINESS_TIME period. To create a product table for a retail store as a bitemporal table, issue the CREATE TABLE statement, as shown in this example:

```
CREATE TABLE product_info
( sku_no       VARCHAR(14) NOT NULL,
  store_id     VARCHAR(19) NOT NULL,
  amt          INTEGER NOT NULL,
  bus_start    DATE NOT NULL,
  bus_end      DATE NOT NULL,
  sys_start    TIMESTAMP(12) NOT NULL
    GENERATED ALWAYS AS ROW BEGIN,
  sys_end      TIMESTAMP(12) NOT NULL
    GENERATED ALWAYS AS ROW END,
  ts_id        TIMESTAMP(12) NOT NULL
    GENERATED ALWAYS AS TRANSACTION START ID,
  PERIOD BUSINESS_TIME (bus_start, bus_end),
  PERIOD SYSTEM_TIME (sys_start, sys_end)
) IN PRD_DMS_SPACE;
```

To create the associated history table, create a table that has the same columns as the bitemporal table, as shown in this example:

```
CREATE TABLE hist_product_info LIKE product_info
```

To create the link to the bitemporal table and the history table, use the ALTER TABLE statement with ADD VERSIONING clause, as shown in this example:

```
ALTER TABLE product_info ADD VERSIONING USE HISTORY TABLE hist_product_info
```

To prevent overlapping business_time periods, create a unique index and specify the BUSINESS_TIME WITHOUT OVERLAPS clause, as shown in this example:

```
CREATE UNIQUE INDEX product_ix
  ON product_info ( sku_no, store_id, BUSINESS_TIME WITHOUT OVERLAPS)
```

Inserting data into bitemporal tables
Use the INSERT statement to add data into bitemporal tables, as shown in this example:

```
INSERT INTO product_info (sku_no, store_id,  amt, bus_start, bus_end )
INSERT INTO product_info (sku_no, store_id,  amt, bus_start, bus_end )
INSERT INTO product_info (sku_no, store_id,  amt, bus_start, bus_end )
INSERT INTO product_info (sku_no, store_id,  amt, bus_start, bus_end )
```

**Table 16** shows the data added to the product_info table with these INSERT statements:

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
<th>Sys_start</th>
<th>Sys_end</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>6.99</td>
<td>'2013-01-01'</td>
<td>'2013-07-01'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>6.99</td>
<td>'2013-07-01'</td>
<td>'2014-01-01'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR_00019</td>
<td>NJ10</td>
<td>7.99</td>
<td>'2013-01-01'</td>
<td>'2014-01-01'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NJ15</td>
<td>5.99</td>
<td>'2013-01-01'</td>
<td>'2014-01-01'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Updating data in bitemporal tables

Use the UPDATE statement to modify data in bitemporal tables in the same manner that you update system-period temporal tables and application-period tables.

To change the start of the BUSINESS_TIME period for CHR_00005 from 1/1/2013 to 4/1/2013, issue an update, as shown in the following example:

```
UPDATE product_info SET bus_start = '2013-04-01' WHERE sku_no = 'CHR_00005'
```

Because there is no time overlap, the existing row is updated and a new row is inserted in the history table. Table 17 shows the row added to the hist_product_info table:

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00005</td>
<td>NJ15</td>
<td>5.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
<td>09/29/2013</td>
</tr>
</tbody>
</table>

Table 17. Updated data in the history table after update operation

To change the price for CHR_00001 in June and July 2013, issue the same update shown in Listing 4:

```
UPDATE product_info FOR PORTION OF BUSINESS_TIME FROM '2013-06-01' TO '2013-08-01'
SET amt = 14.99 WHERE sku_no = ‘CHR_00001’
```

The time overlap causes the splitting of the two existing rows for CHR_00001. The existing rows are updated to reflect the new period of time and two new rows are added to indicate the price for June and July. Table 18 shows the content of the product_info table after this update:

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>6.99</td>
<td>1/1/2013</td>
<td>6/1/2013</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>7.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>14.99</td>
<td>1/1/2013</td>
<td>7/1/2013</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>14.99</td>
<td>7/1/2013</td>
<td>8/1/2013</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NJ15</td>
<td>5.99</td>
<td>1/1/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
</tbody>
</table>

Table 18. Updated data in a bitemporal table

¹: The value of this timestamp column is shown as a date value so that examples are easier to read.
Deleting data from bitemporal tables

Use the DELETE statement with the FOR PORTION OF BUSINESS_TIME clause to remove data from bitemporal tables. Also, the delete operation adds rows to the associated history table with the appropriate system timestamps.

To delete the product with sky_no 'CHR_00001', remove the row from the product_info table, as shown in this example:

```
DELETE FROM product_info FOR PORTION OF BUSINESS_TIME FROM '2013-06-15' TO '2013-08-15' WHERE sku_no = 'CHR_00001'
```

Table 19 shows the updated data in the product_info table after the delete operation:

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>6.99</td>
<td>1/1/2013</td>
<td>6/1/2013</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>7.99</td>
<td>8/15/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>14.99</td>
<td>6/1/2013</td>
<td>6/15/2013</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00005</td>
<td>NJ15</td>
<td>5.99</td>
<td>1/1/2013</td>
<td>4/1/2014</td>
<td>09/30/2013</td>
<td>12/30/9999</td>
<td>09/30/2013</td>
</tr>
</tbody>
</table>

Table 20 shows the added data in the hist_product_info table after the delete operation:

<table>
<thead>
<tr>
<th>SKU_NO</th>
<th>STORE_ID</th>
<th>AMT</th>
<th>BUS_START</th>
<th>BUS_END</th>
<th>SYS_START</th>
<th>SYS_END</th>
<th>TS_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>6.99</td>
<td>1/1/2013</td>
<td>7/1/2013</td>
<td>09/29/2013</td>
<td>09/30/2013</td>
<td>09/29/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>14.99</td>
<td>6/1/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00001</td>
<td>NY01</td>
<td>14.99</td>
<td>7/1/2013</td>
<td>8/1/2013</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
</tr>
<tr>
<td>CHR_00019</td>
<td>NY01</td>
<td>7.99</td>
<td>8/1/2013</td>
<td>1/1/2014</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
<td>09/30/2013</td>
</tr>
</tbody>
</table>

Querying bitemporal tables

Use the SELECT statement to query a bitemporal table to retrieve results for a specified point or period in time in the same manner that you query application-period and system-period temporal tables.

You can include FOR BUSINESS_TIME, FOR SYSTEM_TIME, or both in the FROM clause, as shown in the following examples:

- The following query lists all the rows from the product_info table with a BUSINESS_TIME period that begins before or on 07/15/2013 and ends after 07/15/2013:
  ```sql
  SELECT sku_no, amt, bus_start, bus_end
  FROM product_info
  FOR BUSINESS_TIME AS OF '2013-07-15'
  WHERE sku_no = 'CHR_00001'
  ```
To retrieve also the rows in the history table, use the FOR SYSTEM_TIME AS OF clause instead.

- The following query lists all the rows in the `product_info` table and the `hist_product_info` table for CHR_00001:

```sql
SELECT sku_no, amt, bus_start, bus_end
FROM product_info
FOR SYSTEM_TIME FROM '0001-01-01-00.00.00.00000000' TO '9999-12-30-00.00.00.000000000000'
WHERE sku_no = 'CHR_00001'
```

- The following query combines two clauses to list all the rows from the `product_info` table and the `hist_product_info` table with a BUSINESS_TIME period that begins before or on 07/15/2013 and ends after 07/15/2013 for CHR_00001:

```sql
SELECT sku_no, amt, bus_start, bus_end
FROM product_info
FOR BUSINESS_TIME AS OF '2013-07-15'
FOR SYSTEM_TIME FROM '0001-01-01-00.00.00.00000000' TO '9999-12-30-00.00.00.000000000000'
WHERE sku_no = 'CHR_00001'
```

- The following query lists all the rows in the `product_info` table for CHR_00001 because there is no time period specification:

```sql
SELECT sku_no, amt, bus_start, bus_end
FROM product_info
WHERE sku_no = 'CHR_00001'
```

For more information about using bitemporal tables and examples, see [Bitemporal tables](#) in the DB2 Information Center.

**Conclusion**

In this tutorial, you have been introduced to the following concepts in preparation for the DB2 10.1 fundamentals certification exam 610:

- How to query data from tables.
- How to sort and group data from tables.
- How to insert, update, delete, and merge data from tables.
- How to commit and rollback transactions.
- How to create DB2 SQL procedures and user-defined functions.
- How to call and retrieve results from SQL procedures and user-defined functions.
- How to identify results from an XQuery.
- How to create temporal tables.
- Manage data and query temporal tables.

For additional information to help you prepare for the DB2 10.1 fundamentals certification exam 610, see the [Related topics](#) section.
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 the DB2 10.1 Information Center to find more details about each of the concepts presented in this tutorial.
- Learn additional information about the DB2 10.1 fundamentals certification exam 610 such as details about the exam and objectives, skills that you need, additional test preparation resources, and how to take the sample assessment test.
- Read an overview of the new capabilities in the DB2 10.1 Announcement letter.
- Get a DB2 10.1 trial download for AIX, HP-UX Itanium, Linux for System i, Linux for System p, Linux for System x86, Linux for System x86-64, Linux for System z (64 bit), Windows (32bit), Windows (64bit).
- Download IBM Data Studio at no charge from this developerWorks page that offers full product images for all platforms.

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)