Topic 2.3: Additional SQL Statements and Features

This module covers some of the more important SQL statements and features that were added in recent versions of the SQL Standard, namely SQL:2008 and SQL:2011.

The MERGE Statement

The SQL MERGE statement, which was added to the SQL:2008 standard, is a powerful statement that combines the ability to insert new rows into a table and to update or delete existing rows. SQL programmers often use the term *upsert* (a blended word formed for the words update and insert) when referring to merge operations. The merge is accomplished by comparing the rows in a source table with the rows in the target table. When matches are found, the MERGE statement can specify which columns should be updated in the target table, or that the matching row in the target table should be deleted. When matches are not found, the MERGE statement can specify the insert of the new row in the target table. The best way to understand the MERGE statement is by looking at an example.

The following diagram shows the MERGE_INVENTORY table that contains inventory updates to be applied to the CD_INVENTORY master table. In effect, the merge operation performs a left outer join of the MERGE_INVENTORY and CD_INVENTORY tables, inserting unmatched rows and (in this example) updating matched rows. The state of the CD_INVENTORY table after the merge completes is shown at the bottom of the diagram with new/changed values shaded in grey.

MERGE_INVENTORY

MERGE_CD_NAME: VARCHAR(60)	MERGE_MUSIC_TYPE: VARCHAR(30)	MERGE_PUBLISHER: VARCHAR(50)	MERGE_IN_STOCK: INT
Different Shades of Blue	Blues	J&R Adventures	42
Come On Over	Country	Mercury	6
Innervisions (Remastered)	R&B	Motown / Universal	18
The Definitive Collection	R&B	Motown / Universal	34
Man Against Machine	<null>	Warner Bros.	8

CD_INVENTORY (Before Merge)

CD_NAME: VARCHAR(60)	MUSIC_TYPE: VARCHAR(30)	PUBLISHER: VARCHAR(50)	IN_STOCK: INT
Drive All Night	Alternative Rock	ANTI Records	12
Different Shades of Blue	<null>	J&R Adventures	42
Innervisions (Remastered)	<null>	Motown / Universal	16
The Definitive Collection	R&B	Independen	34
Rumours (Reissued)	Classic Rock	Warner Bros.	17

CD_INVENTORY (After Merge)

CD_NAME: VARCHAR(60)	MUSIC_TYPE: VARCHAR(30)	PUBLISHER: VARCHAR(50)	IN_STOCK: INT
Drive All Night	Alternative Rock	ANTI Records	12
Different Shades of Blue	Blues	J&R Adventures	42
Innervisions (Remastered)	R&B	Motown / Universal	18
The Definitive Collection	R&B	Motown / Universal	34
Rumours (Reissued)	Classic Rock	Warner Bros.	17
Come On Over	Country	Mercury	6
Man Against Machine	<null>	Warner Bros.	8

Here is the MERGE statement syntax for this example:

```
MERGE INTO CD_INVENTORY
USING MERGE_INVENTORY ON (CD_NAME = MERGE_CD_NAME)
WHEN MATCHED THEN
  UPDATE SET MUSIC_TYPE = MERGE_MUSIC_TYPE,
             PUBLISHER = MERGE_PUBLISHER,
             IN_STOCK   = MERGE_IN_STOCK
WHEN NOT MATCHED THEN
  INSERT (CD_NAME, MUSIC_TYPE, PUBLISHER, IN_STOCK)
    VALUES (MERGE_CD_NAME, MERGE_MUSIC_TYPE,
          MERGE_PUBLISHER, MERGE_IN_STOCK);
```

Temporal Features

Temporal data is data that changes over time. Long-awaited temporal features were added to the SQL:2011 standard. The standard supports three types of temporal tables: system-versioned tables, application-time period tables, and system-versioned application-time period tables (a combination of the other two types).

Being relatively new, vendor adoption of these new features is a work in progress. IBM's DB2 Version 10 supports all three table types. Oracle 12*c* and Teradata 13.10 and 14 support system-version tables, but not the other types. As of this writing, MySQL and SQL Server do not yet support SQL temporal features.

System-Versioned Tables

A *system-versioned table* is a table that contains current rows as well as row history that is automatically maintained by the DBMS from a system (database) time perspective. System-versioned tables are useful in situations where an accurate history of data changes must be maintained for business and/or legal reasons. For example, a financial institution must keep previous versions of customer account information so that customers, auditors, and regulatory agencies can be provided a detailed history of account information. With system-versioned tables, previous versions of table rows are automatically preserved whenever an update or delete of a row is successfully processed by the DBMS. Furthermore, in order to maintain the integrity of the row history, only the DBMS is allowed to maintain the data values in the period begin and period end columns for each version of a row.

Here is the CREATE TABLE statement for a system-versioned table named EMPLOYEE:

```
CREATE TABLE EMPLOYEE
(EMP_ID          INT                  PRIMARY KEY NOT NULL,
```

```
EMP_NAME      VARCHAR(50)       NOT NULL,
SYS_BEGIN     TIMESTAMP(12)     GENERATED ALWAYS AS ROW BEGIN NOT NULL,
SYS_END       TIMESTAMP(12)     GENERATED ALWAYS AS ROW END   NOT NULL,
PERIOD FOR SYSTEM_TIME (SYS_BEGIN, SYS_END)
)
WITH SYSTEM VERSIONING;
```

As you can see, the table must be defined with columns that hold dates or timestamps for the beginning and ending effective time period. The PERIOD FOR SYSTEM TIME clause tells the DBMS that this is a system-versioned table and provides the names of the effective period columns. Vendor implementations may have somewhat different syntax – always check your vendor documentation.

Once created, you can use ordinary SQL to maintain the data in the table, but with one important restriction: only the DBMS can update the effective time period dates. The DBMS manages these dates automatically, so the effective time period columns are never mentioned in an SQL INSERT, UPDATE or DELETE statement.

As an example, here are some inserts, an update, and a delete for the EMPLOYEE table, followed by a listing of the table contents after the statements were run.

```
INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME)
    VALUES (1, 'Kim I. Payne');

INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME)
    VALUES (2, 'Benjamin R. Vazquez');

INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME)
    VALUES (3, 'Agatha U. Lee');

UPDATE EMPLOYEE
  SET EMP_NAME = 'Agatha L. Chang'
WHERE EMP_ID   = 3;

DELETE FROM EMPLOYEE
WHERE EMP_ID = 2;
```

Query results:

```
EMP_ID   EMP_NAME           SYS_BEGIN             SYS_END
------   ------------------ ------------------    -----------------

   1     Kim I. Payne       2015-03-22 01:24:04   9999-12-30 00:00:00
   3     Agatha L. Chang    2015-03-22 12:20:48   9999-12-30 00:00:00
```

As you can see, ordinary SQL statements see only the current data in the table. Previous values for updated columns or deleted rows are not visible. This is as it should be, but there are cases when someone needs to see the historic data. SQL:2011 provides a new FOR SYSTEM TIME clause that can be used to specify a point in time or a span of time for which the rows are to be displayed. Here is a statement that displays the entire history for the data in the EMPLOYEE table, followed by the result from the query.

```
SELECT EMP_ID, EMP_NAME, SYS_BEGIN, SYS_END
FROM EMPLOYEE
  FOR SYSTEM_TIME FROM TIMESTAMP '0001-01-01 00:00:00'
              TO TIMESTAMP '9999-12-31 23:59:59'
ORDER BY EMP_ID, SYS_BEGIN;
```

Query results:

```
EMP_ID   EMP_NAME            SYS_BEGIN             SYS_END
------   -------------------   -------------------   ------------------

   1     Kim I. Payne          2015-03-22 01:24:04   9999-12-30 00:00:00
   2     Benjamin R. Vazquez   2015-03-22 01:24:04   2015-03-22 12:20:48
   3     Agatha U. Lee         2015-03-22 01:24:04   2015-03-22 12:20:48
   3     Agatha L. Chang       2015-03-22 12:20:48   9999-12-30 00:00:00
```

Note that you can see both the old and new versions of the row that was updated (EMP_ID 3) and that the deleted row (EMP_ID 2) is visible as well.

Application-Time Period Tables

An *application-time period table* is a table that contains rows with effective time periods that are assigned by the database user rather than the database system. Rows of data can have time periods in the past, present or the future. However, row history is not rigorously maintained as it is with system-versioned tables and period being and end dates can be directly updated by the database user. Application-time period tables are most useful in situations where business application requirements call for capturing time periods when table rows are considered effective in the real world. For example, an insurance application must keep track of policy changes over time, so that when a claim occurs, it can be processed against the policy as it existed on the date of the event that precipitated the need for the claim, such as the date of an auto accident or the date of a visit to a medical office.

For example, here is the CREATE TABLE statement for an application-time period table named CD_PROMOTION, which holds promotional pricing information for music CDs.

```
CREATE TABLE CD_PROMOTION
  (CD_ID             INT              NOT NULL,
  SALE_PRICE         NUMERIC(5,2)     NOT NULL,
```

```
  BEGIN_EFF_DT          DATE                NOT NULL,
  END_EFF_DT            DATE                NOT NULL,
  PERIOD FOR BUSINESS_TIME (BEGIN_EFF_DT, END_EFF_DT),
  PRIMARY KEY (CD_ID, BUSINESS_TIME WITHOUT OVERLAPS)
);
```

Note that as with the system-versioned table example, we have beginning and ending effective date (or timestamp) columns for the time period, and the PERIOD FOR clause tells the DBMS that these are the period effective date columns. The lack of a WITH SYSTEM VERSIONING clause tells the DBMS that this is an application-time period table instead of a system-versioned table. Also note that the PRIMARY KEY includes the time period (BUSINESS_TIME in this example) and specifies WITHOUT OVERLAPS to prevent overlapping time periods. Per the SQL standard, the time period name can be any name except SYSTEM_TIME, but DB2 allows only the name BUSINESS_TIME.

Unlike system-versioned tables, the period effective date values must be supplied by the database user in any SQL statement that inserts or updates rows of data. Dates can also be updated if they were incorrectly entered.

Here is a series of SQL statements and a listing of the CD_PROMOTION table data the resulted from these statements.

```
INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (101, 5.39, '2015-01-01', '2015-01-15');

INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (102, 10.79, '2015-03-01', '2015-03-31');

INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (102, 10.50, '2015-03-31', '2015-04-15');

INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (103, 9.00, '2015-01-01', '2015-05-01');

INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (104, 5.00, '2017-01-01', '9999-12-31');
```

```
UPDATE CD_PROMOTION
  SET BEGIN_EFF_DT = '2016-01-01'
WHERE CD_ID = 104;
```

Query Results:

```
CD_ID      SALE_PRICE      BEGIN_EFF_DT      END_EFF_DT
-----      ----------      ------------      ----------
101         5.39            2015-01-01        2015-01-15
102        10.79            2015-03-01        2015-03-31
102        10.50            2015-03-31        2015-04-15
103         9.00            2015-01-01        2015-05-01
104         5.00            2016-01-01         9999-12-31
```

Given the above data, suppose you want the sale price for CD 103 to be 8.50 for the period of 4/1/2015 to 4/15/2015 If you tried to insert a new row using the following INSERT statement, you would receive an error because the time period would overlap with the existing row for CD 103 that has a time period of 1/1/2015 to 5/1/2015:

```
INSERT INTO CD_PROMOTION
    (CD_ID, SALE_PRICE, BEGIN_EFF_DT, END_EFF_DT)
VALUES (103, 8.50, '2015-04-01', '2015-04-15');
```

You could solve this by deleting the existing row and inserting two new rows, each with the appropriate dates and sale price. However, SQL provides the FOR PORTION OF clause that allows an update that is effective for a desired span of time. The DBMS will split the existing row into two or three rows with the correct effective dates and other values. Here is an example:

```
UPDATE CD_PROMOTION
  FOR PORTION OF BUSINESS_TIME FROM '2015-04-01' TO '2015-04-15'
  SET SALE_PRICE = 8.50
WHERE CD_ID = 103;
```

Query Results:

```
CD_ID      SALE_PRICE      BEGIN_EFF_DT      END_EFF_DT
-----      ----------      ------------      ----------
 101        5.39            2015-01-01        2015-01-15
 102       10.79            2015-03-01        2015-03-31
 102       10.50            2015-03-31        2015-04-15
 103        9.00            2015-01-01        2015-04-01
 103        8.50            2015-04-01        2015-04-15
 103        9.00            2015-04-15        2015-05-01
 104        5.00            2016-01-01        9999-12-31
```

You can use the normal DELETE statement to completely remove all rows that match the WHERE clause in the statement. Moreover, you can use the FOR PORTION OF clause in a DELETE statement to split an existing record. In the following example, the row for CD_ID 101 is deleted and the 104 is split in two because of a delete of a portion of its time span.

```
DELETE FROM CD_PROMOTION
WHERE CD_ID = 101;

DELETE FROM CD_PROMOTION
  FOR PORTION OF BUSINESS_TIME FROM '2016-03-01' TO '2016-03-31'
WHERE CD_ID = 104;
```

Query Results:

CD_ID	SALE_PRICE	BEGIN_EFF_DT	END_EFF_DT
102	10.79	2015-03-01	2015-03-31
102	10.50	2015-03-31	2015-04-15
103	9.00	2015-01-01	2015-04-01
103	8.50	2015-04-01	2015-04-15
103	9.00	2015-04-15	2015-05-01
104	5.00	2016-01-01	2016-03-01
104	5.00	2016-03-31	9999-12-31

System- Versioned Application-Time Period Tables

As the name suggests, a *system-versioned application-time period table* combines the features of system-versioned and application-time period tables into a single structure with effective time periods (application time periods) assigned by the database user in addition to row change history with system time periods assigned by the database system. In recognition of the two different time periods that are tracked for each row of data, many practitioners use the term *bitemporal* when referring to system-versioned application-time period tables. Bitemporal tables are useful in situations where business application requirements call for capturing time periods when rows are considered effective in the real world combined with audit or regulatory requirements which call for rigorous tracking of all changes made to rows of data in the table. For example, a financial account management application must keep track of the effective fiscal period for each transaction as well as maintain an audit trail of all changes to the financial records.

The following CREATE TABLE statement creates a system-versioned application-time period table called CD_PROMOTION_HISTORY.

```
CREATE TABLE CD_PROMOTION_HISTORY
(CD_ID          INT                NOT NULL,
```

```
SALE_PRICE      NUMERIC(5,2)      NOT NULL,
BEGIN_EFF_DT    DATE              NOT NULL,
END_EFF_DT      DATE              NOT NULL,
SYS_BEGIN       TIMESTAMP(12)    GENERATED ALWAYS AS ROW BEGIN NOT NULL,
SYS_END         TIMESTAMP(12)    GENERATED ALWAYS AS ROW END   NOT NULL,
PERIOD FOR BUSINESS_TIME (BEGIN_EFF_DT, END_EFF_DT),
PERIOD FOR SYSTEM_TIME (SYS_BEGIN, SYS_END)
PRIMARY KEY (CD_ID, BUSINESS_TIME WITHOUT OVERLAPS)
)
WITH SYSTEM VERSIONING;
```

The database user is responsible for supplying effective date values for the application time period (BEGIN_EFF_DT and END_EFF_DT in this example), but the only the DBMS can manage the date values for the system time period (SYS_BEGIN and SYS_END in this example). As with application-time period tables, the FOR PORTION OF clause can be used on DELETE and UPDATE statements.

Source: *SQL, A Beginner's Guide*, 4th Edition, Andrew J. Oppel, McGraw-Hill Education, 2015