Database Systems Lab

SQL DML

After SQL DDL, let's review the most important statements of SQL DML. This includes instructions for querying (`SELECT`), inserting (`INSERT`), modifying (`UPDATE`), and deleting (`DELETE`) rows in tables. It also includes the `MERGE` statement, which allows us to insert, update, and delete rows in a table at the same time, but we don't deal with that, and you won't need it during the test either (if you're interested, check the documentation).

We already know the `SELECT` statement, we won't deal with that now either. Some people classify `SELECT` into a separate sublanguage, the Data Query Language (DQL), which includes this single statement. However, since querying is an operation on rows of tables, it can also be considered part of the DML.

There are two basic forms of the `INSERT` statement. With one of them, we can insert one row at a time into a table; the syntax of this is as follows:

```
INSERT INTO <table_name> [(<column_name>[,...])]
    VALUES (<expression>[,...])
```

`<table_name>` is the name of the table where we want to insert a new row. Then we can specify column names in parentheses separated by commas. If we do so, we assign values only to the specified columns in the expression list following the `VALUES` keyword, in the order in which the column names are listed. Columns in the table that are not listed take their default values or, if no default value is specified, `NULL`. (Remember the `DEFAULT` keyword in the `CREATE TABLE` and `ALTER TABLE` statements?) If we do not specify a column list, we give values for all columns in the order specified in the table definition. If we click on a table in SQL Developer, the structure (columns) of the table will appear on the right, where, in addition to the names, types, and default values of the columns, we can also see a `COLUMN_ID` column. This tells you the ordering of the columns in the table.

The `VALUES` keyword is followed in parentheses by expressions whose values are assigned to the specified columns or to all columns. (Note that these expressions can be subqueries if they result in a single-column and single-row table.) The type of expressions you specify must be convertible to the appropriate column type. If this is not true, or the new row violates an integrity constraint (for example, assigning a `NULL` value to a column that cannot take a `NULL` value, or a key has a value that already exists in the table), you receive an error message, and the new row is not inserted.

Using the other form of the `INSERT` statement, we can insert multiple rows at once:

```
INSERT INTO <table_name> [(<column_name>[,...])]
    SELECT ...
```

In this case, the rows obtained as a result of the query are inserted into the table. Of course, the number of columns in the `SELECT` list must match the number of columns specified or, if no column list is specified, the number of columns in the table. The type of each column in the query must be convertible to the type of the corresponding column in the table. If any of the rows we want to insert violates an integrity constraint, we will receive an error message, and no rows will be inserted. If no rows are returned by the query, the table will not change.

There are also two forms of the `UPDATE` statement. The syntax of one of them is as follows:

```
UPDATE <table_name> SET <column_name> = <expression>[,...]
    [WHERE <condition>]
```

<table_name> is the name of the table to be updated. <column_name> specifies a column we want to update, and <expression> defines the new value for that column in each modified row (the expression can also be given by a subquery, of course). We can list several of such column/expression pairs, separated by commas. If there is a WHERE, only rows that satisfy the condition are updated, if there is no WHERE, then all rows in the table.

If we change values in multiple columns at once, in some cases it may be easier to specify the new values for all columns with a single subquery. Then we can use the other form of UPDATE:

```
UPDATE <table_name> SET (<column_name>[,...]) = (SELECT ...)
    [WHERE <condition>]
```

In this case, the nested SELECT should have as many columns as the column names listed after the SET keyword, and of course, the type of each column in the query should be convertible to the type of the corresponding column in the table.

As with INSERT, if a new value in a column violates an integrity constraint in any row, we receive an error message, and no rows are changed.

Finally, here is the syntax of the DELETE statement:

```
DELETE [FROM] <table_name> [WHERE <condition>]
```

This statement is quite simple: rows that meet the condition are deleted from the table. If no condition is specified, all rows in the table are deleted (the table itself will not be deleted; it will be an empty table). The FROM keyword is optional.

There is also a TRUNCATE TABLE statement, the simplest form of which looks like this:

```
TRUNCATE TABLE <table_name>
```

This statement also deletes all rows in the specified table, without deleting the table itself. There are two important differences between the unconditional DELETE and the TRUNCATE TABLE statements:

- TRUNCATE TABLE is not part of DML, but part of DDL. This implies that we cannot undo its effect (at least in Oracle; see the ROLLBACK statement later in the chapter titled *SQL DCL*).
- TRUNCATE TABLE is faster than DELETE, especially if indexes, triggers (snippets of code that run automatically in response to a specific activity), or other dependencies are defined on the table, because these are ignored by TRUNCATE TABLE.

That's all about SQL DML. Now, as usual, let's look at some examples of how to use these three statements.

In the first few examples, we use the two tables created in the chapter titled *SQL DDL* as follows:

```
CREATE TABLE owner
(
    id            NUMBER(5) CONSTRAINT owner_pk PRIMARY KEY,
    name          VARCHAR2(30 CHAR) NOT NULL,
    birth_date    DATE,
    ssn           CHAR(11 CHAR) NOT NULL CONSTRAINT owner_uq_ssn UNIQUE
);
```

```
CREATE TABLE car
(
    license_plate_number   VARCHAR2(6 CHAR) CONSTRAINT car_pk PRIMARY KEY,
    color      VARCHAR2(10 CHAR),
    owner_id   NUMBER(5) CONSTRAINT car_fk_owner REFERENCES owner,
    price      NUMBER(8,2) CONSTRAINT car_ch_price CHECK (price > 10000)
);
```

Add a new owner with ID 1, named Adam Taylor, who was born on January 1, 2000, and whose SSN is 30001011234.

```
INSERT INTO owner
    VALUES (1, 'Adam Taylor', DATE'2000-01-01', '30001011234');
```

If we want to specify these data in a different order, we can list the columns in the order we want. For example:

```
INSERT INTO owner (id, birth_date, name, ssn)
    VALUES (1, DATE'2000-01-01', 'Adam Taylor', '30001011234');
```

Add a new owner with ID 1, named Ann Jones, with an SSN of 29901011234, and with an unknown date of birth.

```
INSERT INTO owner (id, birth_date, name, ssn)
    VALUES (1, NULL, 'Ann Jones', '29901011234');
```

We'll get an error message because we're violating the primary key constraint named `owner_pk`.

Add a new owner with ID 2, named Ann Jones, with an SSN of 29901011234, and with an unknown date of birth.

```
INSERT INTO owner (id, birth_date, name, ssn)
    VALUES (2, NULL, 'Ann Jones', '29901011234');
```

The same more briefly:

```
INSERT INTO owner (id, name, ssn)
    VALUES (2, 'Ann Jones', '29901011234');
```

Update the `owner` table so that the default value of the owners' date of birth is the current date.

```
ALTER TABLE owner
    MODIFY birth_date DEFAULT sysdate;
```

Add a new owner with ID 3, named Paul Jones, with an SSN of 19901025678, and with the current date as date of birth.

```
INSERT INTO owner (id, birth_date, name, ssn)
    VALUES (3, sysdate, 'Paul Jones', '19901025678');
```

The same more briefly:

```
INSERT INTO owner (id, name, ssn)
    VALUES (3, 'Paul Jones', '19901025678');
```

Let's check what we've done so far. List the rows of the `owner` table.

```
SELECT * FROM owner;
```

Delete owners whose date of birth is unknown.

```
DELETE FROM owner
    WHERE birth_date IS NULL;
```

Add the owner Ann Jones again with the data provided earlier.

```
INSERT INTO owner (id, birth_date, name, ssn)
    VALUES (2, NULL, 'Ann Jones', '29901011234');
```

Note that the shorter form no longer works since we added a default value to the `birth_date` column.

Update the `owner` table to set the date of birth of owners for whom it is not currently known. Assume that the date of birth can be determined from the SSN: characters 2 and 3 indicate the year of birth, characters 4 and 5 indicate the month, and characters 6 and 7 indicate the day.

```
UPDATE owner
    SET birth_date = to_date(substr(ssn, 2, 6), 'YYMMDD')
    WHERE birth_date IS NULL;
```

Add a new car with a license plate number ABC123, red color, a price of 100000, and an owner of Paul Jones. (We can assume that there is only one owner named Paul Jones.)

```
INSERT INTO car
    VALUES ('ABC123', 'red',
        (SELECT id FROM owner WHERE name = 'Paul Jones'), 100000);
```

Note that it is not a good solution to specify the owner ID as a constant. Remember that all solutions must work on any database.

Add a new car with a license plate number ZZZ999, and with an unknown color, price, and owner.

```
INSERT INTO car (license_plate_number)
    VALUES ('ZZZ999');
```

Update the data of the car with a license plate number ZZZ999: the color should be brown, the price should be 10000, and the owner should be Adam Taylor. (We can assume that there is only one owner named Adam Taylor.)

```
UPDATE car
    SET color = 'brown', price = 10000, owner_id =
        (SELECT id from owner where name = 'Adam Taylor')
    WHERE license_plate_number = 'ZZZ999';
```

With this statement, we violate the constraint on the price of the car, called `car_ch_price`.

Update the data of the car with a license plate number ZZZ999: the color should be brown, the price should be 20000, and the owner should be Adam Taylor. (We can assume that there is only one owner named Adam Taylor.)

```
UPDATE car
    SET color = 'brown', price = 20000, owner_id =
        (SELECT id from owner where name = 'Adam Taylor')
    WHERE license_plate_number = 'ZZZ999';
```

In the following few examples, we will use the `patron` table, which was also created in the chapter titled *SQL DDL*, like this:

```
CREATE TABLE patron AS
    SELECT last_name, first_name, to_char(birth_date, 'YYYY.MM.DD') bdate
        FROM BOOK_LIBRARY.customers;
```

Modify the patrons' first and last names in the `patron` table: append a plus sign (+) to both the first and last name of patrons born after 1999.

```
UPDATE patron SET last_name = last_name || '+',
                  first_name = first_name || '+'
    WHERE bdate LIKE '2%';
```

After executing the statement, we can see that 5 rows were updated.

Add to the `patron` table a new column named `cnt`, which is an integer of up to five digits.

```
ALTER TABLE patron
    ADD cnt NUMBER(5);
```

If we check the contents of the table, we can see that the new column will have a `NULL` value in each row.

Modify the values in the `cnt` column of the `patron` table so that in each row, it contains the number of borrowings for that patron.

```
UPDATE patron SET cnt =
    (SELECT COUNT(b.book_item_id)
        FROM BOOK_LIBRARY.customers c LEFT JOIN BOOK_LIBRARY.borrowing b
            ON library_card_number = customer_id
        WHERE trim(trailing '+' from patron.last_name) = c.last_name
            AND trim(trailing '+' from patron.first_name) = c.first_name
            AND patron.bdate = to_char(c.birth_date, 'YYYY.MM.DD'));
```

This solution will not give a good result if we have multiple patrons with the same name and date of birth. However, because the `patron` table was created without patron IDs, there is no way we can distinguish between such patrons. We also assumed that the original first and last names of patrons do not end with a plus sign. Check the resulting numbers of borrowings.

Delete all rows of the `patron` table.

```
DELETE patron;
```

Repopulate the `patron` table with the same data that was in it before we deleted it.

For this solution, we make use of the last example from the chapter titled *SQL DDL*:

```
INSERT INTO patron
    SELECT
        CASE WHEN extract(year from birth_date) > 1999 THEN
            last_name || '+'
        ELSE
            last_name
        END,
        CASE WHEN extract(year from birth_date) > 1999 THEN
            first_name || '+'
        ELSE
            first_name
        END,
        to_char(birth_date, 'YYYY.MM.DD'),
        COUNT(book_item_id)
        FROM BOOK_LIBRARY.customers LEFT JOIN BOOK_LIBRARY.borrowing
            ON library_card_number = customer_id
        GROUP BY library_card_number, last_name, first_name, birth_date
        ORDER BY last_name, first_name;
```