Database Systems Lab

SQL DDL

Now we will get acquainted with the most important statements of SQL DDL (Data Definition Language). Most of these statements allow us to create (`CREATE`), modify (`ALTER`), or delete (`DROP`) a schema object. For example, we have the `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE` statements for managing tables, or `CREATE VIEW`, `ALTER VIEW`, `DROP VIEW` for managing views. These statements are not discussed at reference level, we highlight only the parts that are more important to us.

The syntax for one form of the `CREATE TABLE` statement is as follows:

```
CREATE TABLE <table_name>
(
    {<column_definition>|<table_constraint>}[,...]
)
```

This statement creates a new, empty table (without rows), we only describe its structure. `<table_name>` will be the name of the new table, it is an identifier. Then column definitions and table constraints are listed in parentheses, separated by commas. `<column_definition>` looks like this:

```
<column_name> <type> [DEFAULT <expression>] [<column_constraint>]...
```

`<column_name>` is an identifier, `<type>` specifies the type of the column, and `<column_constraint>` specifies some restriction for the column. There are many types that can be substituted for `<type>`, but for now it is enough for us to know the following built-in types:

- `NUMBER[(p[,s])]`: Numeric data, that is, a number. It ranges from 10^{-130} to 10^{126}. Its representation is usually fixed-point. `p` specifies the precision of the number, i.e., the maximum number of significant decimal digits. Its value can range from 1 to 38. `s` is the number of digits in the fractional part. Its value can range from –84 to 127. If negative, it specifies how many digits of the whole part should be rounded (for example, –2 rounds to hundreds). If `s` is not specified, it is treated as 0, i.e., it means an integer. If `p` is not specified either, then it represents a floating-point number with maximum range and precision.
- `CHAR[(n [{BYTE|CHAR}])]`: A fixed-size string with a length of `n` bytes or characters. `n` defaults to 1 and has a maximum value of 2000. Strings shorter than `n` are padded with spaces.
- `VARCHAR2(n [{BYTE|CHAR}])`: A variable-size string with a maximum length of `n` bytes or characters. The maximum value of `n` is at least 4000.
- `DATE`: Date and time. Its range extends from 4712 BC to 9999 AD. It stores year, month, day, hour, minute, and second. However, it does not store fractions of seconds or time zone information. Its size is fixed at 7 bytes.

`DEFAULT <expression>` sets the default value for the column if we don't give it an explicit value when we insert a row. If omitted, the default is `NULL`.

`<column_constraint>` can be one of the following:

- `UNIQUE`: The column will be a key of the table, so its value must be unique in each row. However, it can take a `NULL` value on multiple rows. A table can have multiple keys.
- `PRIMARY KEY`: The column becomes the primary key of the table, so its value must be unique across each row, and it cannot take a `NULL` value. A table can have only one primary key (or none).
- `NOT NULL`: The column cannot take a `NULL` value in any row. (You can also specify `NULL`, but this does not imply any constraint; it merely explicitly indicates that a column can take a `NULL` value.)
- `REFERENCES <table_name>[(<column_name>)]`: The column becomes a foreign key of the table, which references the column with the specified name in the specified table. The given column must be a key or primary key of the other table. If we don't specify a column name, it references the primary key of the other table.
- `CHECK (<condition>)`: Specifies a general constraint on the column: the value in the column must satisfy the specified condition in each row, that is, the condition must be either true or `NULL`.

`<table_constraint>` can be one of the following:

- `UNIQUE (<column_name>[,...])`: The specified columns together form a compound key of the table.
- `PRIMARY KEY (<column_name>[,...])`: The specified columns together form a compound primary key of the table.
- `FOREIGN KEY (<column_name>[,...])`
  `REFERENCES <table_name>[(<column_name>[,...])]`: The specified columns together form a compound foreign key of the table that references the columns with the specified names in the specified table. The referenced columns must form a key or primary key of the other table. If we do not specify referenced columns, it will reference the primary key of the other table.
- `CHECK (<condition>)`: Imposes a general constraint on multiple columns at once: the value of the referenced columns must satisfy the specified condition in each row, that is, the condition must be either true or `NULL`.

As you can see, apart from the `NOT NULL` constraint, each constraint can be either a column constraint (if it applies to one column) or a table constraint (if it applies to one or more columns).

Each column and table constraint definition can be supplemented at the beginning with a `CONSTRAINT <constraint_name>` clause that names that constraint. We can refer to it later by this name, for example, if we want to delete it. If we don't specify it, there will still be a name for the constraint, generated by the system.

Another form of the `CREATE TABLE` statement looks like this:

```
CREATE TABLE <table_name>
    AS SELECT ...
```

In this case, a new table is created whose structure and rows are defined by a query. The columns of the table will be named as specified in the `SELECT` list (expressions must be renamed if they are not simple column names), and the column types will also be the types of the expressions in the `SELECT` list. There will be no constraints on the new table. If you want, you can add constraints later by using the `ALTER TABLE` statement. The table will have as many rows as the rows returned by the query, so it may also be an empty table.

The general form of the `ALTER TABLE` statement is as follows:

```
ALTER TABLE <table_name>
    <action>[...]
```

`<action>` can be many things, it is enough for us to know the following:

- `ADD ({<column_definition>|<table_constraint>}[,...])`: Adds a new column or table constraint to the table.
- `MODIFY ({<column_definition>|<constraint>}[,...])`: Changes the properties (type, default value, constraints) of an existing column or the definition of an existing constraint. We can specify a constraint by its name using `CONSTRAINT <constraint_name>`, but we can also specify a primary key or key constraint directly using `PRIMARY KEY` or `UNIQUE (<column_name>[,...])`, respectively.
- `DROP {<columns>|<constraint>}`: Deletes one or more existing columns or an existing constraint. You can delete a single column by specifying `COLUMN <column_name>`. Multiple columns can be deleted by listing them in parentheses (`<column_name>[,...]`). The constraint can be specified in the same forms as for `MODIFY`.
- `RENAME COLUMN <column_name> TO <column_name>`: Renames an existing column.
- `RENAME CONSTRAINT <constraint_name> TO <constraint_name>`: Renames an existing constraint.
- `RENAME TO <table_name>`: Renames the table.

The syntax of the `DROP TABLE` statement is as follows:

```
DROP TABLE <table_name> [CASCADE CONSTRAINTS] [PURGE]
```

The statement deletes the table from the database. The `CASCADE CONSTRAINTS` keyword also deletes any foreign key constraints that reference the table. If it is not specified, and there are foreign keys referencing the table, we receive an error message, and the table is not deleted. The `PURGE` keyword causes the storage space occupied by the table to be irrevocably freed. If it is not specified, the table is only moved to the Recycle Bin (which is a data dictionary table called `RECYCLEBIN`, containing information about deleted objects) and can later be restored from there.

After tables, the next schema object we are discussing is *view*. A view is nothing more than a named query. When we reference it, the query runs on the current database state, and we can work with the result of the query. It is called a view because individual users (or rather roles) see only a certain part of the entire database, and this part is often defined by one or more views. For example, Neptun's student view is the part of Neptun's database that students have access to. But there is also a teacher view, department admin view, faculty admin view, etc.

We can create our own view with the following statement:

```
CREATE VIEW <view_name>
    AS SELECT ...
```

This is the simplest form of the statement. After the view name, we could also specify column names and table constraints in parentheses, but we will not address this. Such a view is useful when we have a relatively complicated query that we often need. In this case, we do not have to type and run the long `SELECT` every time, but just write:

```
SELECT * FROM <view_name>
```

And this will have exactly the same effect as if we had written:

```
SELECT * FROM (SELECT ...)
```

Here, the nested `SELECT` is the query in the view definition.

In many cases, views can not only be queried, but we can also insert new rows, delete rows, or update rows, just like with tables, but only in the case of so-called updatable views. We will not go into this in more detail now.

You can also modify views to some extent:

```
ALTER VIEW <view_name>
    <action>[...]
```

`<action>` can be, but is not limited to, one of the following:

- `ADD`: adds a new constraint to the view.
- `MODIFY CONSTRAINT`: modifies an existing constraint.
- `DROP`: deletes an existing constraint.
- `COMPILE`: recompiles the view.

`COMPILE` can be used, after changing the structure of a table on which the view is based, to verify that the change does not affect the view. This allows us to detect any errors before running the view.

We can delete a view like this:

```
DROP VIEW <view_name> [CASCADE CONSTRAINTS]
```

`CASCADE CONSTRAINTS` does the same thing as for `DROP TABLE`: it also deletes any foreign key constraints that reference the view. If it is not specified, and foreign key references exist for the view, we receive an error message, and the view is not deleted.

I would like to briefly touch on three more schema objects (these will not occur in the test):

- *Index*: Indexes can primarily be created for one or more columns in a table. They are used to provide direct access to rows in a table when a query is based on the specified columns, so the query can be significantly faster than without an index. If an index exists, SQL interpreter will use it automatically. An index is automatically created for the primary key and each key in a table. We can create an index with the following statement (in the simplest case):

  ```
  CREATE INDEX <index_name> ON <table_name> (<column_name>[,...])
  ```

- *Synonym*: This is an alternative name for a schema object. Not all schema objects can have a synonym, but we can create a synonym, for example, for a table, view, or even another synonym. But what is it good for? First, we can give an object with a long or complicated name a shorter or more meaningful name (for example, instead of `BOOK_LIBRARY.book_items`, `instance` might be simpler and more understandable). Second, you can keep previously written applications functional even if you rename certain objects that those applications use. Third, synonyms allow you to move objects to another schema (or even to another database) without having to change your code. We can create a synonym with the following statement:

```
CREATE SYNONYM <synonym> FOR <schema_object>
```

- *Sequence*: Sometimes we can't define an obvious primary key in a table. In this case, what we usually do is add a column of integer type that will serve as the primary key. The value of this column starts at 1 and increments by one for each new row we insert. A sequence is a schema object that generates such a sequence of numbers for us, guaranteeing the uniqueness of the keys (if used correctly). We can specify the first element of the sequence, its increment, direction, minimum and maximum elements, and whether it should be cyclic or finite. Here is an example of creating a sequence:

```
CREATE SEQUENCE key_seq
    START WITH 1000
    INCREMENT BY 10
    MINVALUE 100
    MAXVALUE 9990
    CYCLE;
```

  The first element of this sequence is 1000, the second is 1010, the third is 1020, etc. After 9990, the value of the next element will be 100 (if we had not specified the minimum value, it would have been 1, which is the default minimum value for ascending sequences). The value that follows is 110, etc. Once the sequence has been created, we can use it as follows: the current element of the sequence can be queried using the `currval` pseudocolumn, and the next element can be queried using the `nextval` pseudocolumn. For example:

```
SELECT keys_seq.nextval from dual;
```

  This query gives 1000 on the first run, 1010 on the second run, etc.

Finally, I will mention the `RENAME` statement, which allows us to rename a table, view, synonym, or sequence. The general form of the statement is as follows:

```
RENAME <old_name> TO <new_name>
```

Now, without explanation, let's see some examples to illustrate the statements above.

Create a table named `owner`, in which we want to store car owner data. We need the following columns:

- **`id`: an integer of up to 5 digits, the ID of the owner, primary key of the table.**
- **`name`: a string of up to 30 characters, the name of the owner, cannot take a `NULL` value.**
- **`birth_date`: a value of type date, the birth date of the owner.**
- **`ssn`: a string of exactly 11 characters, the SSN of the owner, key of the table, cannot take a `NULL` value.**

Assign a name to each constraint, except for those regarding `NULL` values.

```
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
);
```

After you create the table, open the list of tables labeled *Tables (Filtered)* under the connection name in SQL Developer. There you should see the new table called `OWNER`. If you don't see it, click the little blue double arrow (Refresh) at the top of the *Connections* window, or press Ctrl+R. If you have it, click on the name `OWNER`. In a new tab on the right, the table will open, and you will see its columns (*Columns* tab). Check the type of columns and which of them can take `NULL` values. Next, click on the *Constraints* tab and check the constraints of the table, especially their names and types.

Create a table called `car`, in which we want to store car data. The following columns should be included in the table:

- **`license_plate_number`: a string of up to 6 characters, the license plate number of the car, primary key of the table.**
- **`color`: a string of up to 10 characters, the color of the car.**
- **`owner_id`: an integer of up to 5 digits, the ID of the car's owner, foreign key of the table, referencing the primary key of the `owner` table.**
- **`price`: a real number with up to 8 significant digits and 2 digits after the decimal point, the price of the car, which must be greater than 10000.**

Name all the constraints.

```
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)
);
```

Alternatively, we could do the same thing by specifying the constraints as table constraints (though it's a bit longer, many people prefer this form):

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

Modify 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;
```

Check the columns of the table on the *Columns* tab. In the row of the `birth_date` column, `sysdate` appeared in the `data_default` column. (The example also shows that if you change only a single column or constraint in the table, you don't have to enclose it in parentheses. The same goes for adding a new column or constraint; see the following example).

Add a new column named `model` to the `car` table, which contains the car model as a string of up to 20 characters.

```
ALTER TABLE car
    ADD model VARCHAR2(20 CHAR);
```

Increase the maximum size of the `model` column of the `car` table to 50 characters.

```
ALTER TABLE car
    MODIFY model VARCHAR2(50 CHAR);
```

Rename the `model` column of the `car` table to `nameplate`.

```
ALTER TABLE car
    RENAME COLUMN model TO nameplate;
```

Rename the constraint on the `car` table named `car_ch_price` to `car_check_price`.

```
ALTER TABLE car
    RENAME CONSTRAINT car_ch_price TO car_check_price;
```

Delete the `car_check_price` constraint on the `car` table.

```
ALTER TABLE car
    DROP CONSTRAINT car_check_price;
```

Add the previously defined `car_ch_price` constraint to the `car` table again.

```
ALTER TABLE car
    ADD CONSTRAINT car_ch_price CHECK (price > 10000);
```

Delete the `nameplate` column of the `car` table.

```
ALTER TABLE car
    DROP COLUMN nameplate;
```

Rename the `car` table to `automobile`.

```
RENAME car TO automobile;
```

Or:

```
ALTER TABLE car RENAME TO automobile;
```

Create a table called `patron` that contains the first and last name and date of birth of our current patrons. The date of birth should be stored as a string in Hungarian format; the name of the column should be `bdate`.

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

Without renaming, this statement would lead to an error.

Add a new constraint to the `patron` table: the primary key of the table should be made up of the three columns together. Also, change the size of the `last_name` and `first_name` columns to 50 characters.

```
ALTER TABLE patron
    ADD PRIMARY KEY (last_name, first_name, bdate)
    MODIFY (last_name VARCHAR2(50 CHAR),
            first_name VARCHAR2(50 CHAR));
```

Create a view called `patrons` that shows the first and last name and date of birth of our patrons. The date of birth should be displayed in Hungarian format; the name of the column should be `bdate`.

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

From the `patrons` view, query the data of patrons born after 1999.

```
SELECT * FROM patrons
    WHERE bdate LIKE '2%';
```

Delete the `patron` table permanently.

```
DROP TABLE patron PURGE;
```

Delete the `owner` table.

```
DROP TABLE owner;
```

Because the `owner` table is referenced by foreign keys, this statement causes an error. One solution is to delete the referring tables first (this is now `car`), and the other is to delete only the foreign keys. We can do this by using the `ALTER TABLE` statement (separately for each referring table), or we can delete all referencing foreign keys at once like this:

```
DROP TABLE owner CASCADE CONSTRAINTS;
```

Finally, as a matter of curiosity, let's take a look at the following exercise:

Create a table named `patron` that contains the first and last name, date of birth, and the number of borrowings of our current patrons. The column names should be `last_name`, `first_name`, `bdate`, and `no_of_borrowings`, respectively. Also include patrons who have never borrowed a book. Append a plus sign (+) to the end of the first and last name if the patron was born after 1999. Store the date of birth as a string, in Hungarian format. Sort the result by last name and then by first name.

```
CREATE TABLE patron AS
    SELECT
        CASE WHEN extract(year from birth_date) > 1999 THEN
            last_name || '+'
        ELSE
            last_name
        END last_name,
        CASE WHEN extract(year from birth_date) > 1999 THEN
            first_name || '+'
        ELSE
            first_name
        END first_name,
        to_char(birth_date, 'YYYY.MM.DD') bdate,
        COUNT(book_item_id) no_of_borrowings
        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;
```

The `CASE` operator works similarly to the conditional operator (`?:`) in C (or Java), but you can specify any number of conditions (`WHEN` branches). The result of this operation is the expression that follows the `THEN` keyword in the first `WHEN` branch that evaluates to true. If there are no such conditions, the expression after `ELSE` will be the result (or `NULL` if there is no `ELSE`). (The `CASE` operator can also function similarly to the `switch` operator in Java, i.e., it can compare the value of a specified expression to different values.)

After running the statement, check the contents of the table. Note the plus signs after the first and last name of young patrons. It is also worth checking the type of each column, which were determined by the interpreter based on the specified expressions. For example, for strings, the maximum length in the column type matches the length of the longest string. In the case of numbers, the type is plain `NUMBER`, i.e., a real number (even if the number of borrowings is always an integer).