SQL Avanzado

CI-0127 Bases de Datos, Universidad de Costa Rica

Sivana Hamer

Importante: Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

1 NULL

In SQL, if the column does not have a *NOT NULL* constraint, we can have a *NULL* value.

- A NULL value can mean that the value is unknown, value withheld or it is not applicable. However, we do not know the type of NULL as we save the same NULL value for all cases.
- As we could compare NULL values, SQL uses a three-valued logic (TRUE, FALSE or UNKNOWN) instead of boolean logic (TRUE or FALSE). Thus, the used truth tables in SQL are shown in Tables 1, 2 and 3.

	TRUE	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Figure 1: AND truth table

	TRUE	**FALSE**	**UNKNOWN**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Figure 2: OR truth table

TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Figure 3: NOT truth table

- While **SELECT**ing tuples in a **WHERE** clause, only tuples that are TRUE are selected. There is an exception for outer joins.
- To compare nulls we use the **IS** command. The following example shows how it is used, to find all the *CARNET*s that have *NULL IMAGE*s.

```
SELECT CARNET
FROM IDENTIFICATION
WHERE IMAGE IS NULL;
```

2 Nested

We can *nest* the result of a query to be used in another query. The outside query is called the *outer query*, while inside the query is the *innermost query*. When a nested query return a single attribute and tuple, we can use a = comparator. In any other case, we shall use the **IN** to compare a value within a set of values and returns multiple tuples. In the following example, we can see how we can get all the *COURSE*s *NAME*s and *CREDITS* that have a *GROUP* in the first *SEMESTER* of the *YEAR* 2022.

```
SELECT NAME, CREDITS
FROM COURSE
WHERE ACRONYM IN (
        SELECT ACRONYM
        FROM _GROUP
        WHERE SEMESTER=1 AND YEAR = 2022
);
```

We can also compare multiple values using *tuples* as $(column_1, column_2, \cdots column_n)$ in the outer query's **WHERE** clause. There are other comparison operators that can be used, as shown in Table 4.

Operator	Description
IN, ANY, SOME	Returns the tuples if we find a value IN the set of values of the inner query.
ALL	With a comparison operator, returns the tuples that are more, less, equal, ... than ALL the innermost tuples.
EXISTS	Returns the tuples if tuples EXIST in the innermost query.
NOT EXIST	Returns the tuples if tuples does NOT EXIST in the innermost query.
NOT EXIST	Returns TRUE if there are no duplicate tuples in the innermost query.

Figure 4: Possible compators with nested queries with multiple tuples

- We can use an outer query attribute in an innermost query, we must use an **ALIAS**. The ALIAS can also be used to reduce ambiguities if both tables have the same name.
- Nested queries are *correlated* if the innermost query references an attribute of the outermost query.
- Nested queries that have a SELECT-FROM-WHERE that use = or **IN** can always be written as a single block query.

3 JOIN

We can **JOIN** tables (can be more than two) together in the **FROM** clause of a **SELECT** operation. We can see in the following example how we can specify a join, to find the *GROUP*'s information for those given in the first *SEMESTER* of the *YEAR* 2022.

There are different types of JOINs. We shall assume that the relation R is joined to S, such that R is the leftmost relation and S the rightmost one.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C JOIN, _GROUP AS G ON C.ACRONYM = G.ACRONYM
WHERE SEMESTER = 1 AND YEAR = 2022;
```

- **INNER JOIN or JOIN**: JOINs relations R and S on a pair of attributes that must have the same values. Each pair of attributes is included only once in the resulting relation.
- **LEFT OUTER JOIN OR LEFT JOIN**: JOINs the relations R and S, ensuring that even if R does not have a matching tuple in S, the tuple in R will appear in the resulting relation with NULL values for S's attributes.
- **RIGHT OUTER JOIN OR RIGHT JOIN**: JOINs the relations R and S, ensuring that even if S does not have a matching tuple in R, the tuple in S will appear in the resulting relation with NULL values for R's attributes.
- **FULL OUTER JOIN OR FULL JOIN**: JOINs the relations R and S, ensuring that even if R or S does not have a matching tuple in the other relation, the tuple will appear in the resulting relation with NULL values.
- **NATURAL <TYPE> JOIN:** A JOIN with no condition specified as the pair of attributes with the same name in R and S are joined once in the resulting relation. The type can be blank (for INNER JOIN), LEFT, RIGHT or FULL. This is similar to an EQUIJOIN in relational algebra.

4 Aggregate functions

With *aggregate functions* information from several tuples can be summarized into one. NULL values are discarded in aggregate functions except for **COUNT(*)**.

Function	**Description**
COUNT	COUNTs the number of tuples in a query.
SUM	SUMs the values of a column in a query (must be numeric).
MAX	Gets the MAX value of a column in a query (must be numeric).
MIN	Gets the MIN value of a column in a query (must be numeric).
AVG	Gets the AVG of the values of a column in a query (must be numeric).

Figure 5: Aggregate functions

There are several ways we can use these functions. For example, we can gather the SUM, MAX, MIN and AVG of *CREDITS* for *COURSE*s.

```
SELECT MAX(CREDITS), MIN(CREDITS), SUM(CREDITS),
AVG(CREDITS) AS AVG_CREDITS
FROM COURSE;
```

We can also count the number of current *COURSE*s saved in the database with the following query.

```
SELECT COUNT(*)
FROM COURSE;
```

5 More clauses SELECT

The full select clause has the following clauses (the order of execution is detailed with the elevated number).

```
SELECT⁶ <attribute list>
FROM¹ <table list>
WHERE² <condition>
GROUP BY³ <grouping condition>
HAVING⁴ <group condition>
ORDER BY⁵ <attribute list>;
```

Thus, the description for the two new clauses (GROUP BY and HAVING) is the following:

- **GROUP BY:** Groups tuples based on a grouping condition. The attributes within the grouping attribute must appear in the SELECT clause, with the desired aggregate functions. A separate GROUP is created for NULL values in a condition.
- **HAVING:** Details a condition to filter groups. It is different from the WHERE clause that filters tuples, while the HAVING clause removes groups.

We can see an example of the **GROUP BY** clauses in the following example. Here, we add an *AREA_NUMBER* to *COURSE*s that has the AREA in which the course is given. For example, computer science courses are within the engineering area. Thus, we group all the *COURSE*s by the area and then gather this number, the number of courses given by area and the average number of *CREDITS*.

```
SELECT AREA_NUMBER, COUNT(*), AVG (CREDITS)
FROM COURSE
GROUP BY AREA_NUMBER;
```

The following example shows how to filter with the **HAVING** clause. This adds from the previous example, where now the *AREA_NUMBER* that have an average less than or equal to 3 are filtered from the results.

```
SELECT AREA_NUMBER, COUNT(*), AVG (CREDITS)
FROM COURSE
GROUP BY AREA_NUMBER
HAVING AVG (CREDITS) > 3;
```

6 Assertions

General constraints can be specified using *declarative assertions* using the **CREATE ASSERTION** statement. The following shows an example of an assertion named *POSITIVE_CREDITS* that checks that no *COURSE*s have non-positive *CREDIT*s.

```
CREATE ASSERTION POSITIVE_CREDITS
CHECK ( NOT EXISTS ( SELECT *
                     FROM COURSE
                     WHERE CREDITS < 1));
```

- This constraint is only checked when tuples are inserted or updated in a specific table.

- **CREATE ASSERTION** should only be used when it is not possible to define a **CHECK**.
- The technique to write an ASSERTION is to define a query that selects all the tuples that would violate the condition, by using the **NOT EXISTS** clause.
- The form of an assertion is:

```
CREATE ASSERTION <name>
CHECK (<query that selects all tuples that violates a condition>);
```

- We cannot define this type of constraints in SQL Server.

7 Triggers

Triggers [1] are constraints that are used to *monitor updates, maintain consistency of derived data and update derived data* the database. We can create this constraint with the **CREATE TRIGGER** command. Triggers have three components:

- **EVENTS (E):** The events (e.g., INSERTs, UPDATEs, DELETEs, temporal period time) that activate the trigger. The trigger can be activated **BEFORE**, **INSTEAD OF**, or **AFTER** the operation that activates the trigger is executed.
- **CONDITION (C):** Determines if the action rule shall be executed if a condition is true, detailed in the **WHERE** clause. This component is optional and if it is not defined, the action will be executed once the event is activated.
- **ACTION (A):** SQL statements that are executed when the trigger is activated.

If a trigger has the optional **FOR EACH ROW** clause, then it is activated separately for each tuple and is known as a *row-level trigger*. If not, it will be activated once per statement and be known as a *statement-level trigger*.

8 Views

A *view* [2] is a table that is defined from other tables (*base tables*), creating a *virtual table* that is not physically stored in the database. Thus there are limitations to updating views, but not querying views. The tables used in the view are called the *defining tables*. The following is an example of a view named *COURSE_GROUPS* that saves the *NAME*, *ACRONYM* and number of groups of *COURSE*s for each *COURSE*.

```
CREATE VIEW COURSE_GROUPS (NAME, ACRONYM, NUMBER_GROUPS)
AS SELECT NAME, G.ACRONYM, COUNT(*)
FROM COURSE AS C JOIN _GROUP AS G ON C.ACRONYM = G.ACRONYM
GROUP BY NAME, G.ACRONYM;
```

We can execute queries over views. We can see an example where all the course names and number of groups are retrieved for the view.

```
SELECT NAME, NUMBER_GROUPS
FROM COURSE_GROUPS;
```

[1] https://docs.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers
[2] https://docs.microsoft.com/en-us/sql/relational-databases/views/views

- If none of the view attributes are results of aggregate functions, the view name attributes do not need to be specified as they are inherited.
- Views simplify certain queries and serve as security or authentication mechanisms. With the latter, only certain access can be given to attributes.
- As views have to be up-to-date, they are materialized when a query is specified to the view.
- We can remove views with the **DROP VIEW** command.
- Usually, views data can be modified if it only has only one table. This is the case for SQL Server [3].
- Within SSMS, we can see all the views in *Databases* > *Views*, as shown in Fig. 6.

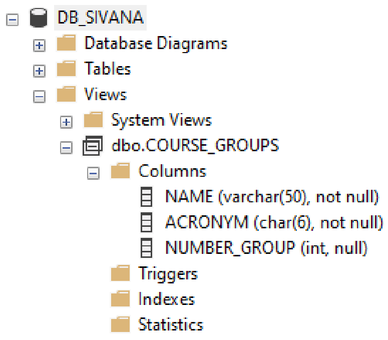
```
DB_SIVANA
  Database Diagrams
  Tables
  Views
    System Views
    dbo.COURSE_GROUPS
      Columns
        NAME (varchar(50), not null)
        ACRONYM (char(6), not null)
        NUMBER_GROUP (int, null)
      Triggers
      Indexes
      Statistics
```

Figure 6: Views in SSMS

9 Stored procedures

With the previous commands, there is an assumption that queries run in the application server. However, we can execute the program in the database server using stored procedures [4] or functions [5] (*persistent stored modules*).

In the following example, we can see how a stored procedure is defined that receives a *COURSE*s' *ACRONYM* and gives the *NAME*.

```
CREATE PROCEDURE GetNameCourse (@ACRONYM CHAR(6))
AS BEGIN
        SELECT NAME
        FROM COURSE
        WHERE ACRONYM = @ACRONYM
END;
```

We can call persistent stored modules using the **EXECUTE** or **EXEC** command, as follows.

```
EXECUTE GetNameCourse @ACRONYM = 'CI0127'
```

- Parameters have a name, a type (SQL data types) and a mode. The mode is IN (input only), OUT (output only), INOUT (both input and output). Thus, if a parameter has IN it can receive the parameter, while OUT indicates that it returns it.

[3] https://docs.microsoft.com/en-us/sql/relational-databases/views/modify-data-through-a-view

[4] https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine

[5] https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions

- The difference with both of these modules is that functions return data, while stored procedures do not.
- These persistent stored modules are stored persistently in the database server and are useful if a database program is needed by several applications (reduces effort and improved modularity), in certain situations reduce data transfer costs between client and server, and enhance modeling power.
- Within the declaration of the persistent stored modules, we can have control-flow statements [6].
- Within SSMS, we can see all the stored procedures and functions in *Databases > Programmability > Stored Procedures* and *Databases > Programmability > Functions*, respectively. This is shown as shown in Fig. 7.

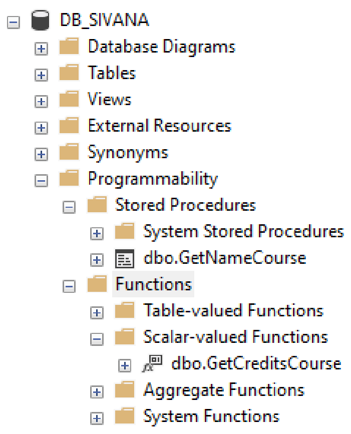

Figure 7: Persistent stored modules in SSMS

References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016, chapters 7, 10.4 and 26.1.

[2] Microsoft. Microsoft sql documentation. [Online]. Available: https://docs.microsoft.com/en-us/sql

[6] https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow