SQL Básico

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 Structured Query Language

The Structured Query Language (SQL) represents a practical relational model for relational DBMS (RDBMS). As there are slight differences with how different RDBMS implement SQL. SQL Server Management Studio (SSMS) as it is the IDE for SQL Server (our RDBMS).

- It defines *what* the results are, not *how* to get them.
- It is based on *relational calculus.*
- It is not the standard language for RDBMS.
- It was defined as a standard by the American National Standards Institute (ANSI) and the International Standard Organization (ISO).
- It defines a language for both DDL and DML.

2 Data definition

The relationship between the terms of the relational model and SQL are shown in Fig. 7.

Relational model	SQL
Relation	Table
Tuple	Row
Attribute	Column

Figure 1: Relationship between the terms of the relational model and SQL

Within SQL, there are different structures that represent the database, as shown in Fig. 2. In the following subsections, each of these levels are detailed.

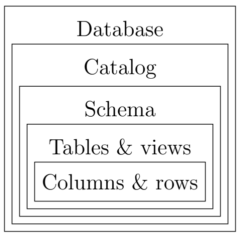


Figure 2: Abstraction levels of database concepts

Database

We can **CREATE** an instance of a database [1] with:

$$\boxed{\textbf{CREATE DATABASE}\ \text{DB\_SIVANA;}}$$

To use that database, we use:

$$\boxed{\textbf{USE}\ \text{DB\_SIVANA;}}$$

We can also add the authorized users with:

$$\boxed{\textbf{CREATE DATABASE}\ \text{DB\_SIVANA}\ \textbf{AUTHORIZATION}\ \text{`sivana';}}$$

Within SSMS, we can see all the databases within the *Object Explorer* > *Databases*, as shown in Fig. 3.

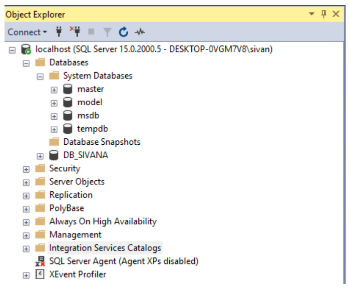


Figure 3: Databases in SSMS

Catalog

A catalog is a collection of schemas [2]. All the schemas within the catalog are saved in the *INFORMATION_SCHEMA*. We can find this schema in *Database* > *Security* > *Schemas* (Fig. 4).

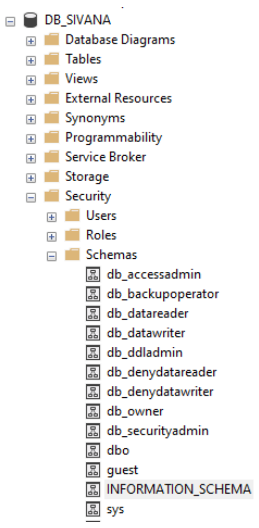


Figure 4: INFORMATION_SCHEMA in SSMS

[1] `https://docs.microsoft.com/en-us/sql/t-sql/statements/create-database-transact-sql`

[2] `https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/catalog-views-transact-sql`

Schema

An SQL Schema represents a group of tables or views that belong to the same database application. It has a:

- Name: Way to identify the Schema
- Authorization identifier: Indicates the user who owns the schema.
- Descriptors: To define the elements within the schema.

We can **CREATE** the schema [3] with:

CREATE SCHEMA UNIVERSITY;

Or we can also add who is authorized by:

CREATE SCHEMA UNIVERSITY **AUTHORIZATION** 'sivana';

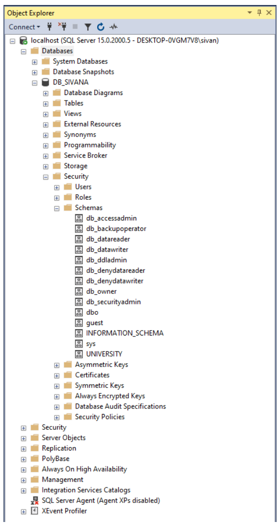
We can later add the descriptors by **ALTER**ing the schema [4] and delete the schema by **DROP**ping it [5]. We can determine the current schema with:

SELECT SCHEMA_NAME();

When we log in into the database installation, we automatically connect to the default catalog and schema. We can change the default schema with:

ALTER USER 'sivana' **WITH DEFAULT_SCHEMA**=UNIVERSITY;

We can find the schemas in *Database* > *Security* > *Schemas* (Fig. 5).



```
localhost (SQL Server 15.0.2000.5 - DESKTOP-0VGM7V8\sivan)
  Databases
    System Databases
    Database Snapshots
    DB_SIVANA
      Database Diagrams
      Tables
      Views
      External Resources
      Synonyms
      Programmability
      Service Broker
      Storage
      Security
        Users
        Roles
        Schemas
          db_accessadmin
          db_backupoperator
          db_datareader
          db_datawriter
          db_ddladmin
          db_denydatareader
          db_denydatawriter
          db_owner
          db_securityadmin
          dbo
          guest
          INFORMATION_SCHEMA
          sys
          UNIVERSITY
        Asymmetric Keys
        Certificates
        Symmetric Keys
        Always Encrypted Keys
        Database Audit Specifications
        Security Policies
  Security
  Server Objects
  Replication
  PolyBase
  Always On High Availability
  Management
  Integration Services Catalogs
  SQL Server Agent (Agent XPs disabled)
  XEvent Profiler
```

Figure 5: Schemas in SSMS

[3] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-schema-transact-sql
[4] https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-schema-transact-sql
[5] https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-schema-transact-sql

Table

A table defines a relation with its name, attributes and initial constraints (Not NULL, keys [6], uniqueness and checks [7]). We can **CREATE** [8] the table with:

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    CREDITS INT NOT NULL,
    UNIQUE (NAME)
);
```

Implicitly, as we are within the default schema (dbo), while we create the table it executes the query as if it was:

```
CREATE TABLE dbo.COURSE;
```

We can see the result of creating the table in *Database* > *Tables* (Fig. 6). There are different types of tables for SSMS [9]. If we expand a table, we can also see its columns, keys, constraints, triggers, indexes and statistics. Double clicking on any of these elements will show more details for each of them.

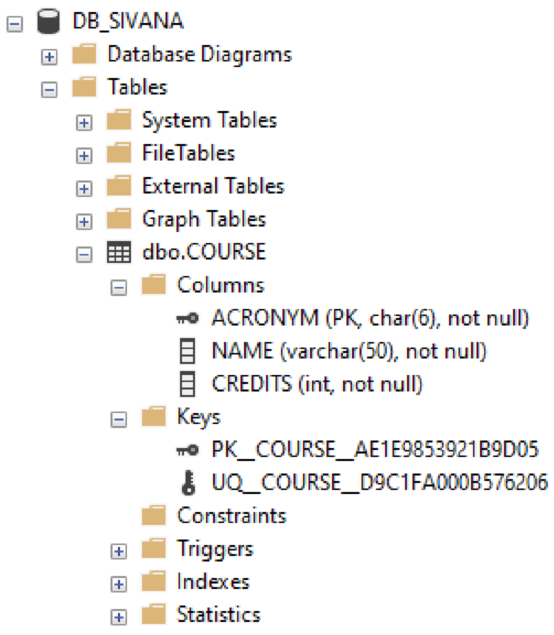


Figure 6: Schemas in SSMS

[6]https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints
[7]https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constrai
[8]https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql
[9]https://docs.microsoft.com/en-us/sql/relational-databases/tables/tables

We can CREATE a foreign key as:

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    NUMBER INT,
    SEMESTER INT,
    YEAR INT,
    PRIMARY KEY (ACRONYM, NUMBER, SEMESTER, YEAR),
    FOREIGN KEY (ACRONYM) REFERENCES COURSE(ACRONYM)
);
```

To define a foreign key, the table must already exist. Therefore, if there is a foreign key fk_1 for table T_1 that depends on a foreign key fk_2 for table T_2 such that fk_1 references T_2 and fk_2 references T_1. Then, the restriction can be defined by **ALTER**ing [10] the table. If we define a column as a primary key, it will automatically add the *NOT NULL* constraint.

3 Data types

The following data types can be created, as shown in Fig. 7.

Category	Data type	Description	Types
Numeric	Integers	An integer number	INT or INTEGER SMALLINT
	Real	A real number	FLOAT or REAL DOUBLE PRECISION
	Formated	Formated number, where i is the number of decimal digits and j the decimal digits after the decimal point	DECIMAL/DEC/NUMERIC(i, j)
Chars	Fixed length	Fixes a char string to have n chars. If shorter, it is padded with spaces to the right.	CHAR(n)
	Variable length	Strings can have a variable length of maximum n chars	VARCHAR(n)
	Large object	Specifies large texts like documents, where n is a size and s the size type (K,M,G)	CLOB(ns)
Bits	Fixed length	Fixes a bit string to have n bits.	CHAR(n)
	Variable length	Strings can have a variable length of maximum n bits.	BIT VARYING(n)
	Large object	Specifies large bit string, where n is a size and s the size type (K,M,G)	BLOB(ns)
	Boolean	Boolean that can be either TRUE or FALSE	BIT
Dates	Date	Given in 'YYYY-MM-DD'.	DATE
	Time	Given in 'HH-MM-SS'.	TIME

Figure 7: SQL data types

[10]`https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql`

Some details with regards to the data types:

- Strings are case sensitive.
- Strings are declared with apostrophes (e.g., 'Sivana').
- We can concatenate strings with || (e.g., 'gallo ' || 'pinto').
- We declare bits with a B preceeding the digits within apostrophes (e.g., B'101').
- For dates, we can also use timezones with TIME WITH TIME ZONE. Without it, the default is the local time zone for the SQL session.
- Some implementations of SQL include more or less types [11] (e.g., SQL does not have an explicit BOOLEAN data type).

4 Constraints

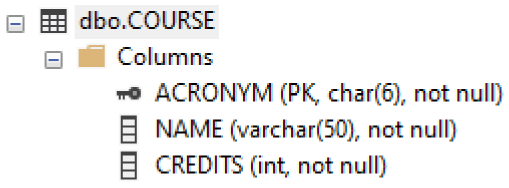
We can define several constraints while creating tables. In the following subsections, different commands are explained.

NOT NULL

For an attribute, we can define a **NOT NULL** constraint to not allow NULL values. This is always implicitly declared for the primary key of the relation. We can see an example of a **NOT NULL** constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    CREDITS INT NOT NULL,
    ...
);
```

We can see that we can define for the columns *NAME* and *CREDITS* the constraint after specifying the name and data type. For the primary key, it is not necessary to specify the constraint as it is done implicitly. We can see within SSMS the restriction for the columns with the constraint in *Database > Tables > Table > Columns > Column*, as shown in Fig. 8.



```
dbo.COURSE
  Columns
    ACRONYM (PK, char(6), not null)
    NAME (varchar(50), not null)
    CREDITS (int, not null)
```

Figure 8: NOT NULL constraint in SSMS

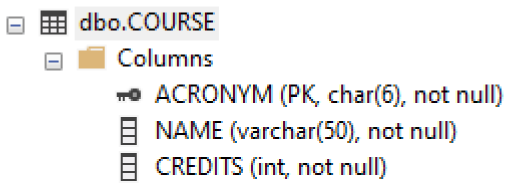
DEFAULT

A default value can be set for a column, with the **DEFAULT** command. If no default is set for a column, the default is *NULL* for all the columns without the **NOT NULL** constraint. We can see an example of this constraint for *COURSE*.

[11] https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

```
CREATE TABLE COURSE (
    ...
    CREDITS INT NOT NULL DEFAULT 4,
    ...
);
```

We set that the *CREDITS* column will have as a default value a four. We can see within SSMS the restriction for the columns with the constraint in *Database* > *Tables* > *Table* > *Constraints*, as shown in Fig. 9.



```
dbo.COURSE
    Columns
        ACRONYM (PK, char(6), not null)
        NAME (varchar(50), not null)
        CREDITS (int, not null)
```

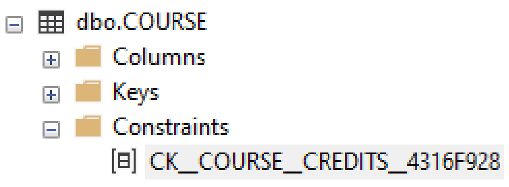
Figure 9: DEFAULT constraint in SSMS

CHECK

We can use a **CHECK** constraint to define a domain, row-based and general constraints. We can see an example of a domain constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ...
    CREDITS INT NOT NULL DEFAULT 4,
    CHECK (CREDITS > 0 AND CREDITS < 13),
    ...
);
```

With this constraint, a *CREDITS* for a *COURSE* has to be $0 < CREDITS < 13$. We can see within SSMS the restriction in *Database* > *Tables* > *Table* > *Constraints*, as shown in Fig. 10.



```
dbo.COURSE
    Columns
    Keys
    Constraints
        CK__COURSE__CREDITS__4316F928
```

Figure 10: CHECK constraint in SSMS

The constraint can also be defined in the following form:

```
CREATE DOMAIN CREDITS AS INT
CHECK (CREDITS > 0 AND CREDITS < 13);
```

We can also define a tuple constraint, where each row that is inserted or updated is checked individually. For example, if we add for _**GROUP** columns for the number of spaces available (*NUMBER_AVAILABLE_SPACES*) and the number of enrolled students (*NUMBER_ENROLLED*). Thus, we can define a row-based constraint to not allow more students enrolled than the available spaces as follows:

```
CHECK (NUMBER_ENROLLED <= NUMBER_AVAILABLE_SPACES);
```

PRIMARY KEY

We can define a **PRIMARY KEY** constraint while creating the column for the table. We can see an example of this constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY KEY,
    ...
);
```

We can also define a primary key of several columns after defining the columns, as seen in the following example for *GROUP*.

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    NUMBER INT,
    SEMESTER INT,
    YEAR INT,
    PRIMARY KEY (ACRONYM, NUMBER, SEMESTER, YEAR),
    ...
);
```

We can see within SSMS the restriction in *Table* > *Columns* and *Table* > *Keys*, as shown in Fig. 11.

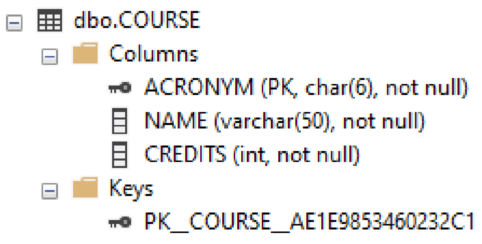

dbo.COURSE
Columns
ACRONYM (PK, char(6), not null)
NAME (varchar(50), not null)
CREDITS (int, not null)
Keys
PK__COURSE__AE1E9853460232C1

Figure 11: PRIMARY KEY constraint in SSMS

UNIQUE

The **UNIQUE** clause defines candidate keys that are unique for the table. We can see an example of this constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ...
    NAME VARCHAR(50) NOT NULL UNIQUE,
    ...
);
```

A **UNIQUE** constraint can also be defined for a group of columns. This can be defined with the following structure, putting the list of columns inside the parentheses:

```
CREATE TABLE COURSE (
    NAME VARCHAR(50) NOT NULL,
    UNIQUE (NAME),
    ...
);
```

We can see within SSMS the restriction in *Table > Keys*, as shown in Fig. 12.

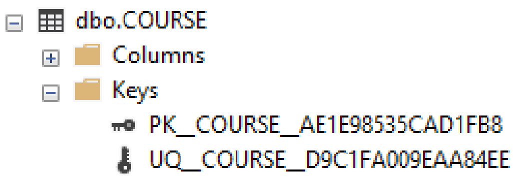

dbo.COURSE
Columns
Keys
PK__COURSE__AE1E98535CAD1FB8
UQ__COURSE__D9C1FA009EAA84EE

Figure 12: UNIQUE constraint in SSMS

FOREIGN KEY

A **FOREIGN KEY** constraint can be defined for a table. This constraint ensures that the constraint is followed while tuples are added or deleted, or when a foreign key or primary key is updated. We can see an example of this constraint for *GROUP*.

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    FOREIGN KEY (ACRONYM) REFERENCES COURSE(ACRONYM)
    ...
);
```

We can see within SSMS the restriction in *Table > Columns* and *Table > Keys*, as shown in Fig. 13.

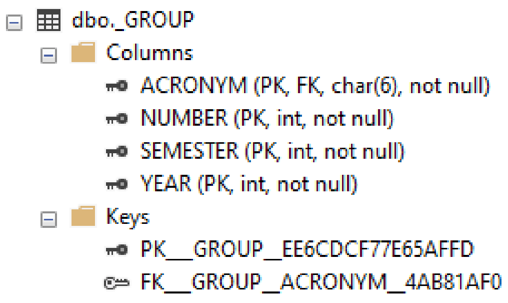

dbo._GROUP
Columns
ACRONYM (PK, FK, char(6), not null)
NUMBER (PK, int, not null)
SEMESTER (PK, int, not null)
YEAR (PK, int, not null)
Keys
PK___GROUP__EE6CDCF77E65AFFD
FK___GROUP__ACRONYM__4AB81AF0

Figure 13: FOREIGN KEY constraint in SSMS

We can define different types of foreign key constraints. The *referential trigger actions* defines the behavior during changes to the foreign key:

- **NO ACTION:** An error is raised and the state of the database is rolled back. This is also known as the **RESTRICT** option and is the default behavior.
- **CASCADE:** When a parent table updates or deletes a foreign key, the reference table *updates* the foreign key. This does not work if the referenced or foreign key is a **TIMESTAMP**.
- **SET NULL:** When a parent table updates or deletes a foreign key, the reference table *sets to NULL* the foreign key.

- **SET DEFAULT:** When a parent table updates or deletes a foreign key, the reference table *sets to the default values* the foreign key. For this constraint to work, all columns of the foreign key must have a default. If a column is nullable without an explicit default value, the default value is NULL.

These actions can be combined for different **FOREIGN KEY** restrictions. Furthermore, restrictions can fire secondary cascading chains and can be subsequently repeated. If the **SET NULL**, **SET DEFAULT** or **CASCADE** action triggers a table with **NO ACTION**, all the updates are stopped and rolled back. These behaviors can be defined for updates (**ON UPDATE**) or deletes (**ON DELETE**). We can see how this works in the following example:

```
CREATE TABLE _GROUP (
    ...
    FOREIGN KEY (ACRONYM) REFERENCES COURSE(ACRONYM)
        ON DELETE NO ACTION ON UPDATE NO ACTION,
    ...
);
```

CONSTRAINT

For every constraint, we can define a name by specifying the keyword **CONSTRAINT**, then the name of the constraint and then the constraint to be named. We can see an example for the *COURSE* table.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6),
    CONSTRAINT PK_ACRONYM_COURSE
        PRIMARY KEY(ACRONYM),
    ...
);
```

We can see within SSMS the constraint with the name, for this example, in *Table > Keys*, as shown in Fig. 14.

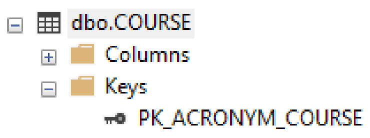


Figure 14: Naming a PRIMARY KEY constraint in SSMS

5 Data modification

There are three ways we can modify the database: inserts, deletes and updates. In the following subsections, each subtype is described and defined.

INSERT

We can **INSERT** [12] into a table rows. It is possible to bulk insert rows [13]. The block of commands are the following:

[12] `https://docs.microsoft.com/en-us/sql/t-sql/statements/insert-transact-sql`
[13] `https://docs.microsoft.com/en-us/sql/t-sql/statements/bulk-insert-transact-sql`

```
INSERT INTO <table (<attribute list>)>
VALUES <value list>;
```

- **INSERT INTO:** Details the table that rows will be inserted. A list of attributes can be given inside the parentheses to define the order of the columns, but is optional.
- **VALUES:** Details the values that will be inserted. The order is given either implicitly (defined by the order in which the table was defined) or explicitly (defined by the order given in the attribute list).

The first way to insert is to detail the table and the values for the columns, in the same way it was defined. For example, here we can see the result of inserting a row into *COURSE*.

```
INSERT INTO COURSE
VALUES ('CI0127', 'Bases de Datos', 4);
```

We can also **INSERT** by explicitly mentioning the order of the columns while inserting. It is necessary to specify all the columns that cannot be NULL and have no default. If a column can be NULL or has a default and is not specified, it will automatically add a NULL or the default value, respectively. This explicit **INSERT** is shown in the following example:

```
INSERT INTO COURSE (NAME, CREDITS, ACRONYM)
VALUES ('Ingeniería de Software', 4, 'CI0126');
```

DELETE

We can **DELETE** [14] delete current rows of a table. This may activate constraints. The full block of commands are the following:

```
DELETE FROM <table>
WHERE <condition>
```

- **DELETE FROM:** Defines the table that rows are deleted from.
- **WHERE:** Details a boolean condition that identifies the rows that will be deleted. This clause is optional.

We can delete all the rows in a table with the following command:

```
DELETE FROM COURSE;
```

DELETing rows does not remove the table definition in the database. To delete a table, we must **DROP** it as follows:

```
DROP COURSE;
```

To **DELETE** only certain rows from a table, we can define a *condition* using the **WHERE** clause.

[14] `https://docs.microsoft.com/en-us/sql/t-sql/statements/delete-transact-sql`

```
DELETE FROM COURSE
WHERE CREDITS > 5 ;
```

UPDATE

We can also **UPDATE** [15] column values for rows. This command is similar to **DELETE**, but has a **SET** clause that defines the values of the updated columns. The full block of commands are the following:

```
UPDATE <table>
SEMESTER <list of columns with values>
WHERE <condition>
```

- **UPDATE:** Defines the table that rows are updated.
- **SET:** Details the list of columns with the values to update them.
- **WHERE:** Details a boolean condition that identifies the rows that will be updated. This clause is optional.

The following is an example that updates all rows in the table:

```
UPDATE COURSE
SET CREDITS = 5;
```

We also define that only rows that meets a *condition* will be updated with the **WHERE** clause, as follows:

```
UPDATE COURSE
SET CREDITS = 10
WHERE ACRONYM = 'CI0127';
```

We can also modify the current value of a column, using the **SET** clause as follows:

```
UPDATE COURSE
SET CREDITS = CREDITS + 1;
```

6 Data retrieval

To view the rows that are in the database, we can use the **SELECT** [16] command. This command has multiple clauses, with the only required clauses being **SELECT** and **FROM**. The block of commands are the following (we will see some more in advanced SQL):

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>
ORDER BY <attribute list>;
```

- **SELECT:** Details the list of columns that will be retrieved from the tables. All the columns can be retrieved with *.

[15] `https://docs.microsoft.com/en-us/sql/t-sql/queries/update-transact-sql`
[16] `https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql`

- **FROM:** Details all the tables needed for the query.
- **WHERE:** Details a boolean condition that identifies the rows that will be retrieved. This clause is optional.
- **ORDER BY:** Defines the order of the rows retrieved. This clause is optional.

We can define a simple query to select all the rows from a table that meet a condition. This is shown in the following example where all the *COURSE*s' *ACRONYM*s for *COURSE*s with more than three *CREDITS* are gathered. To compare values within the **WHERE** clause we use comparison operators. The comparison operators grammar for SQL and the respective math representation are shown in Fig. 15.

```
SELECT ACRONYM
FROM COURSE
WHERE CREDITS > 3;
```

Math	SQL
$=$	=
$\neq$	<>
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=

Figure 15: Comparison operators relationship between math and SQL

We can also define a **SELECT** without a **WHERE** clause to select all the rows in a table as shown in the following example. Here, we gather all the *ACRONYM*s for every row in *COURSE*.

```
SELECT ACRONYM
FROM COURSE;
```

We desire to get all the columns that we can after an operation. One way to do this is to specify all the columns manually. We could also use * in the **SELECT** clause, specifying that all the columns will be retrieved. In the following example, the attributes *ACRONYM*, *NAME* and *CREDITS* are gathered for all the rows in *COURSE* that have more than three credits.

```
SELECT *
FROM COURSE
WHERE CREDITS > 3;
```

A range of values in the **WHERE** condition can be defined with a **BETWEEN**. In the following example, only *COURSE*s with *CREDITS* between three and seven are retrieved.

```
SELECT *
FROM COURSE
WHERE (CREDITS BETWEEN 3 AND 7);
```

In the **FROM** clause we can put more than one table as a list of tables. In the following example, we retrieve the *combinations* of *COURSE*s and *GROUP*s, gathering the *NAME*, *CREDITS*, *NUMBER*, *SEMESTER* and *YEAR*. The combinations retrieved have no relationship if the tables reference one another, they are all the possible combinations between both tables.

```
SELECT NAME, CREDITS, NUMBER, SEMESTER, YEAR
FROM COURSE, _GROUP
```

Implicitly, if there are no attributes that have the same name within the list of tables it treats the column as if it said "Table.Column". In our current example query, an implicit column is *NAME* that is treated as *COURSE.NAME*. If there are two columns with the same name in the list of tables, we must specify *explicitly* this reference.

If we want to specify the tables based on a condition between both tables, we can do it with a *join condition*. Thus, in the **WHERE** clause we specify the column in a table that relates with another table. In the following example we can see the join condition for "C.ACRONYM = G.ACRONYM". Note that the keyword **AS** creates an alias for the tables that can be used to explicitly state the column.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C, _GROUP AS G
WHERE C.ACRONYM = G.ACRONYM;
```

In a **WHERE** clause we can combine various conditions with **AND**s and **OR**s, following boolean logic. There is also a **NOT** condition. We can see in the following example how we can specify the *join condition*, the *GROUP*'s semester must be equal to one and the *GROUP*'s year must be equal to 2022.

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

This query can be further refined to specify the order of the results with the **ORDER BY** clause. This will order the rows starting from the leftmost element of the list of columns to the rightmost (i.e., order first by the first column, second by the second column and so forth). We can further define the order with **ASC** (ascending order) and **DESC** (descending order). The default order is **ASC**. An example is shown in the following query, where the rows are ordered in ascending order by year and then semester.

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

When we retrieve values from columns, we are getting all the values from all the rows, but not the unique values within the current database state. Thus, to gather the distinct values in the **SELECT** clause the **DISTINCT** keyword is stated before the column name. In the following example, we can see how we can gather the distinct *YEAR* values in the rows of the *GROUP*s.

```
SELECT DISTINCT YEAR
FROM _GROUP;
```

Character string columns can be pattern matched. We can use the % and **LIKE** keywords to match an arbitrary number of zero or more characters. In the following example, only the *COURSE*s with *ACRONYM*s that start with 'CI' (computer science courses) are matched.

```
SELECT *
FROM COURSE
WHERE ACRONYM LIKE 'CI%';
```

```
SELECT ACRONYM
FROM _GROUP
WHERE ACRONYM LIKE 'CI____';
```

We can also specify the specific number of characters to match with an underscore (_) for each character. In the following example, only the *COURSE*s with *ACRONYM*s that start with 'CI' (computer science courses) and proceeded by four blank characters are retrieved.

SET operations can be used with **SELECT** queries. The operations that are defined in SQL are:

- **UNION:** For sets A and B, this operation includes all the elements in either A or B are included in the resulting set.
- **INTERSECTION:** For sets A and B, this operation includes all the elements that are in both A and B are included in the resulting set.
- **EXCEPT:** For sets A and B, this operation includes all the elements that are A without any element that is shared with B.

If we define these operations as they are, they will generate the rows without duplicates. If we add the keyword **ALL** (e.g., **UNION ALL**) duplicates are not eliminated. The tables must be *type-compatible relations*, thys must have all the same attributes in the same order. In the following example, the *GROUP*s that were imparted in the *SEMESTER* 1 and the *YEAR* 2022 are retrieved (this example can also be done with an AND).

```
(SELECT DISTINCT ACRONYM, NUMBER
FROM _GROUP
WHERE SEMESTER = 1)
INTERSECTION
(SELECT DISTINCT ACRONYM, NUMBER
FROM _GROUP
WHERE YEAR = 2022);
```

7 Database permissions

While using a database, different permissions are defined for different users. These permissions can grant access to certain capabilities and data. We can **GRANT** [17] or **REVOKE** [18] permisions for users, based on their respective commands.

References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016, chapters 6.

[17] https://docs.microsoft.com/en-us/sql/t-sql/statements/grant-transact-sql
[18] https://docs.microsoft.com/en-us/sql/t-sql/statements/revoke-transact-sql

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY: McGraw-Hill, 2020, chapter 3.

[3] E. Malinowski and A. Martínez, *Material de Apoyo de Bases de Datos I*, 1st ed., Universidad de Costa Rica, 2018, parte VI.

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