

DATA ANALYSIS BOOTCAMP

SQL

SQL

**SQL is the single most requested language for positions of Data Analyst and Data Scientist.**

Ahead of Python, ahead of R, ahead of Excel.

Why do you think that is?

SQL

**S**tructured **Q**uery **L**anguage is a standard database language used to create, manage and query data from relational databases.

We need to clarify some of these words today

* Database
* Relational
* Query

DATABASES

Organised collection of data structured in ways that impose some rules on the data according to needs, e.g.

* Size
* Accuracy
* Security
* Redundancy
* Accessibility
* Governance

We can do a comparison with Excel to understand these needs

TYPES OF DATABASES OPERATIONAL VS ANALYTICAL

**Operational databases** are the backbone of many companies, organisations, and institutions throughout the world today. This type of database is primarily used to **collect, modify, and maintain data** on a day-to-day basis. The type of **data stored is dynamic**, meaning that it changes constantly and always reflects up-to-the-minute information.

**Analytical database** stores and tracks **historical and time-dependent data**. It is a valuable asset for tracking trends, viewing statistical data over a long period, or making tactical or strategic business projections. The type of **data stored is static**, meaning that the data is never (or very rarely) modified, although new data might often be added.

**Data Squad | SQL**

WHAT IS A RELATIONAL DATABASE

**A relational database is a database structured to recognise relations between stored items of information.**

* Relational databases are built of tables, which establish the relation between **tuples (records or rows)** and **attributes (fields or columns).**

* Each table can be related to each other using the concepts of **keys**

RDBMS

A **relational database management system** (RDBMS) is a software application program you use to create, maintain, modify, and manipulate a relational database.

SQL is the *de facto* universal language to work with data from relational databases so naturally, most RDBMS support SQL.SQL is more than just a means for extracting knowledge from data. It’s also a language for defining the structures that hold data so we can organise relationships in the data.

**MySQL** is one of many RDBMS that support the use of SQL.We will use a graphical user interface to work with MySQL called **MySQL workbench**.

ANATOMY OF A DATABASE

ALL THE COMPONENTS WE WILL TALK ABOUT

**- Tables**

**- Columns**

**- Rows**

**- Keys**

**- Relationships**

**- Views**

**Data Squad | SQL**

TABLES, COLUMNS, ROWS

TABLES

Tables are the **main structures in the database**.

Each table always represents a **single, specific subject**. The order of rows and columns within a table is of absolutely no importance. Every table should contain at least one column—known as a **primary key**—that uniquely identifies each of its rows.

**The subject** that a given table represents is usually **either an object or an event**. When the **subject is an object**, the table represents something that is tangible, such as a person, place, or thing. (i.e.: Pilots, products, machines, students, buildings, and equipment ) When the **subject is an event**, the table represents something that occurs at a given point in time and has characteristics you wish to record. (i.e.: judicial hearings, distributions of funds, lab test results, and geological surveys )

COLUMNS - ATTRIBUTES OF THE OBJECT

Columns represent a characteristic of the subject of the table to which it belongs.

**Columns are the structures that store data.** You can retrieve the data in these columns and then present it as information in almost any configuration imaginable.

ROWS - ENTRIES OF THE OBJECT

A row represents a **unique instance of the subject of a table**. It is composed of the entire set of columns in a table, regardless of whether or not the columns contain any values.

**Data Squad | SQL**

DATA TYPES

WHAT TYPES CAN THE SINGLE PIECE OF DATA BE?

**- Integers** (smallint, int, bigint)

**- Char**

**- Varchar** (variable length char)

**- Text** (unlimited length)

**- Serial (auto incrementing:** smallserial, serial, bigserial)

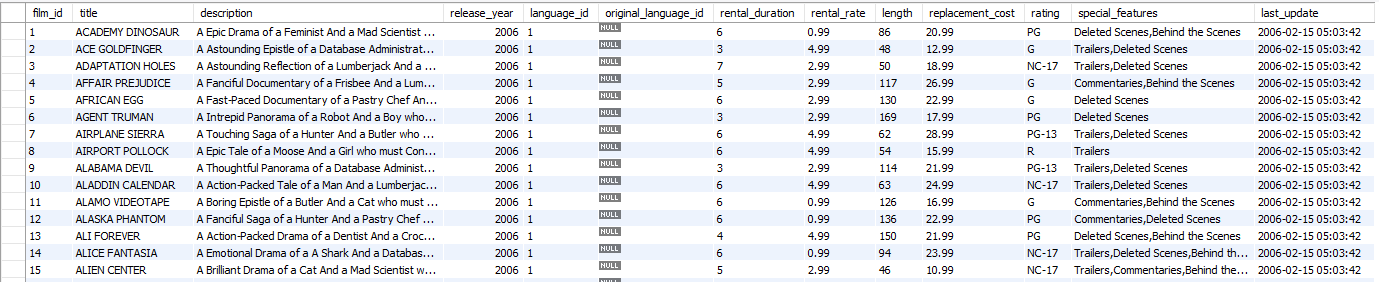
**- Fixed/floating point** (numeric real double precision)

**-Date** (timestamp, date, time, interval)

**Data Squad | SQL**

SAMPLE - SAKILA DATABASE

THE FILMS TABLE



QUERIES I

* Retrieving information from the database
* You have to specify which table(s) you want information for
* What is the criteria for the rows to be returned? any filters?
* How to manipulate the resulting fields?

SELECT STATEMENT

SELECT \* FROM sakila.film;

SELECT title, description, rating FROM sakila.film;

SELECT DISTINCT rental\_duration

FROM sakila.film;

SELECT DISTINCT rental\_duration, language\_id

FROM sakila.film;

**Data Squad | SQL**

ORDER BY

SELECT title, rental\_rate, length

FROM sakila.film

ORDER BY length DESC;

SELECT title, rental\_rate, length

FROM sakila.film

ORDER BY length DESC, rental\_rate DESC;

ALIASING and COMPUTATIONS

SELECT title, rental\_rate AS cost, length

FROM sakila.film;

SELECT title, rental\_rate/length AS price\_per\_min

FROM sakila.film

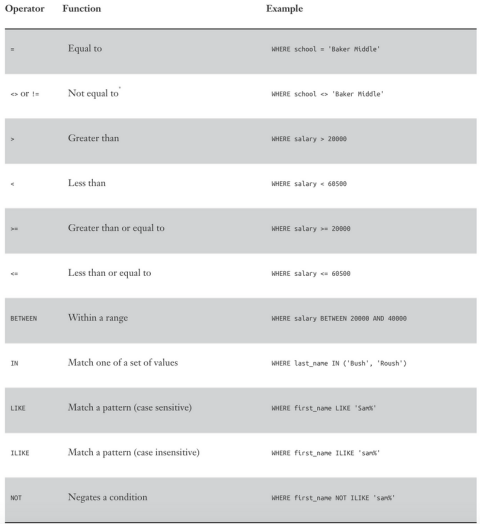
ORDER BY price\_per\_min ASC

SELECT CONCAT(title,', rating:',rating) AS descriptor

FROM sakila.film;

https://www.w3schools.com/sql/sql\_ref\_sqlserver.asp

WHERE - HOW TO RETRIEVE DATA FROM SPECIFIC ROWS



SELECT \*

FROM sakila.film

WHERE rental\_duration = 6;

AGGREGATIONS AND GROUP BY

SELECT COUNT(\*),MAX(rental\_duration),AVG(replacement\_cost),AVG(rental\_duration)

FROM sakila.film

SELECT rating, COUNT(rating), AVG(rental\_rate)

FROM sakila.film

GROUP BY rating;

SELECT rating, rental\_duration, COUNT(rating), AVG(rental\_rate)

FROM sakila.film

GROUP BY rating, rental\_duration;

RELATIONSHIPS

RELATIONSHIPS BETWEEN OBJECTS\TABLES

If rows in a given table can be **associated** in some way with rows in another table, the tables are said to have a relationship between them.

**There are three types of relationships:**

**- 1 to 1**

**- 1 to many**

**- many to many**

**Keys** are a special table construct designed to denote relationships between tables

KEYS

PRIMARY VS FOREIGN

**Every table in your database should have a primary key.** A primary key consists of one column that **uniquely identify** each row within a table.

The primary key:

- identifies a specific row throughout the entire database,

- its column identifies a given table throughout the entire database.

- enforce table-level integrity and help establish relationships with other tables.

When you determine that a pair of tables has a relationship to each other, you typically establish the relationship by taking a copy of the primary key from the first table and inserting it into the second table, where it becomes a **foreign key**.

RELATIONAL DATABASE SCHEMAS

| * Entity Relational Diagram (ERD) * Shows content of each table and connections between tables within a database schema * Primay Key and Foreign Key visual here |  |
| --- | --- |

ONE-TO-ONE RELATIONSHIP

A SINGLE ROW IN THE FIRST TABLE IS RELATED TO ONLY ONE ROW IN THE SECOND TABLE

A pair of tables has a one-to-one relationship when a single row in the first table is related to only one row in the second table, and a single row in the second table is related to only one row in the first table.

In this type of relationship, one table is referred to as the **primary table**, and the other is referred to as the **secondary table**. You establish this relationship by taking the primary key of the primary table and inserting it into the secondary table, where it becomes a **foreign key**. This is a special type of relationship because **in nearly all cases the foreign key also acts as the primary key of the secondary table.**

(e.g. citizen id and passport nr. Notice not all citizens have a passport)

ONE-TO-MANY RELATIONSHIP

ONE ROW IN ONE TABLE HAS A RELATIONSHIP WITH MULTIPLE ROWS IN ANOTHER TABLE

When a pair of tables has a one-to-many relationship, a single row in the first table can be related to many rows in the second table, but a single row in the second table can be related to only one row in the first table.

This relationship is established by taking the primary key of the table on the “one” side and inserting it into the table on the “many” side, where it becomes a foreign key. In this case **the second table has also its own primary key**.

(e.g. city and country - world database)

MANY-TO-MANY RELATIONSHIP

MANY ROWS IN ONE TABLE HAVE A RELATIONSHIP WITH MULTIPLE ROWS IN ANOTHER TABLE

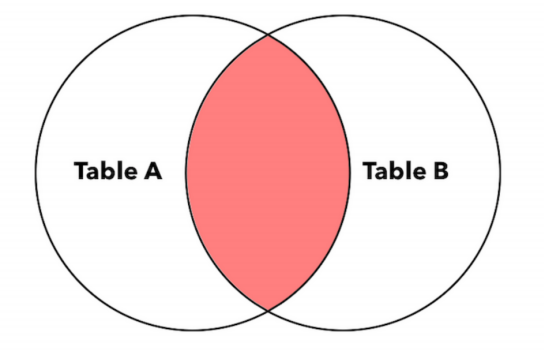
A pair of tables is in a many-to-many relationship when a single row in the first table can be related to many rows in the second table, and a single row in the second table can be related to many rows in the first table.

To establish this relationship properly, you typically create what is known as a **linking table**. This table provides an easy way to associate rows from one table with those of the other and will help to ensure that you have no problems adding, deleting, or modifying any related data. You define a linking table by taking a copy of **the primary key of each table in the relationship and using them to form the structure of the new table**. These columns actually serve two distinct roles: Together they form the composite primary key of the linking table, and separately they each serve as a foreign key.

(e.g. actor and film - sakila)

INNER JOIN

RETURNS ROWS FROM BOTH TABLES WHERE MATCHING VALUES ARE FOUND IN THE JOINED COLUMNS OF BOTH TABLES.

**SELECT** \* 

**FROM** Table\_A

**INNER JOIN** Table\_B

**ON** Table\_A.Key = Table\_B.Key

**SELECT** world.country.Name AS Country\_Name, world.city.Name AS City\_Name, world.country.GNP/world.city.Population AS GNP\_per\_capita

**FROM** world.country

**INNER JOIN** world.city

**ON** world.country.Code **=** world.city.CountryCode;

**Data Squad | SQL**

LEFT JOIN

RETURNS EVERY ROW FROM THE LEFT TABLE PLUS ROWS THAT MATCH VALUES IN THE JOINED COLUMN FROM THE RIGHT TABLE.

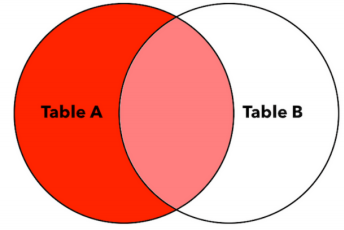
**SELECT** \*

**FROM** Table\_A

**LEFT JOIN** Table\_B

**ON** Table\_A.Key = Table\_B.Key

If there’s no match for a value in the left table, the query result contains NULL values for the right table columns.



RIGHT JOIN

RETURNS EVERY ROW FROM THE RIGHT TABLE PLUS ROWS THAT MATCH THE KEY VALUES IN THE KEY COLUMN FROM THE LEFT TABLE

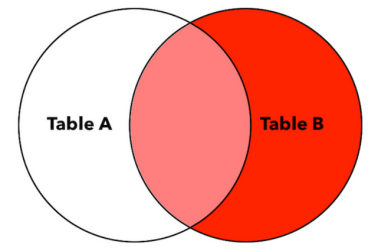
**SELECT** \*

**FROM** Table\_A

**RIGHT JOIN** Table\_B

**ON** Table\_A.Key = Table\_B.Key

If there’s no match for a value in the right table, the query result contains NULL values for the left table columns.



FULL OUTER JOIN

RETURNS EVERY ROW FROM BOTH TABLES AND MATCHES ROWS; THEN JOINS THE ROWS WHERE VALUES IN THE JOINED COLUMNS MATCH.

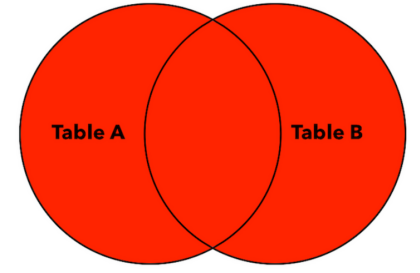
**SELECT** \*

**FROM** Table\_A

**OUTER JOIN** Table\_B

**ON** Table\_A.Key = Table\_B.Key

If there’s no match for a value in either the left or right table, the query result contains an empty row for the other table.



SUBQUERIES

As the name implies, subqueries are queries nested inside another query.

You can use a subquery in the place of any table of the main query, since SELECT statements themselves return “tables”

This is often a solution used for quick analysis since subqueries can easily become very complex

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

**SELECT AVG**(length) **AS** average **FROM** films

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

**SELECT \***

**FROM** sakila.film

**WHERE** length>= **(SELECT AVG**(length) **AS** average **FROM** film)

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

**SELECT** film\_id

**FROM** sakila.film

**WHERE** length> **(SELECT AVG**(length) **AS** average **FROM** film)

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

**SELECT DISTINCT** actor\_id

**FROM** sakila.film\_actor

INNER JOIN (**SELECT** film\_id **FROM** sakila.film **WHERE** length>= **(SELECT AVG**(length) **AS** average **FROM** film)) **AS** selected\_films\_id

**ON** sakila.film\_actor.film\_id = selected\_films\_id.film\_id

SUBQUERIES   
Example: find the names of the actors which starred in movies with lengths higher or equal to the average length of all the movies

**SELECT** first\_name, last\_name

**FROM** sakila.actor

**INNER JOIN** (

**SELECT DISTINCT** actor\_id

**FROM** sakila.film\_actor

INNER JOIN (**SELECT** film\_id **FROM** sakila.film **WHERE** length> **(SELECT AVG**(length) **AS** average **FROM** film)) **AS** selected\_films\_id

**ON** sakila.film\_actor.film\_id = selected\_films\_id.film\_id) **AS** selected\_actors

**ON** sakila.actor.actor\_id = selected\_actors.actor\_id

TEMPORARY TABLES COMPLEXITY BREEDS ERROR

Subqueries can easily become very complex and therefore hard to debug and get wrong

Temporary Tables can be reused and stored for a whole session

They are destroyed after the session, no actual database space is used

TEMPORARY TABLES

**CREATE TEMPORARY TABLE** sakila.selected\_films

**SELECT** film\_id

**FROM** sakila.film

**WHERE** length **> (SELECT AVG**(length) **AS** average **FROM** film**);**

**CREATE TEMPORARY TABLE** sakila.selected\_actors

**SELECT DISTINCT** actor\_id

**FROM** sakila.film\_actor

**INNER JOIN** sakila.selected\_film**s**

**ON** sakila.film\_actor.film\_id = selected\_films.film\_id;

**SELECT** first\_name, last\_name

**FROM** sakila.actor

**INNER JOIN** sakila.selected\_actors

**ON** sakila.actor.actor\_id = selected\_actors.actor\_id;