**SQL MODE INTERMEDIATE TUTORIAL**

[**Aggregation functions** 2](#_Toc119765787)

[SQL Count 2](#_Toc119765788)

[**SQL SUM** 3](#_Toc119765789)

[**SQL MIN/MAX** 3](#_Toc119765790)

[**SQL AVERAGE** 3](#_Toc119765791)

[**SQL GROUP BY** 4](#_Toc119765792)

[SQL HAVING 5](#_Toc119765793)

[**SQL CASE** 5](#_Toc119765794)

[**CASE WITH AGGREGATE FUNCTIONS** 6](#_Toc119765795)

[**USING CASE INSIDE OF AGGREGATION FUNCTIONS** 7](#_Toc119765796)

[**SQL DISTINCT** 8](#_Toc119765797)

[**DISTINCT IN AGGREGATION** 9](#_Toc119765798)

[**SQL JOINS** 9](#_Toc119765799)

[**INNER JOIN** 10](#_Toc119765800)

[**JOINING TABLES WITH IDENTICAL COLUMN NAMES** 10](#_Toc119765801)

[**SQL OUTER JOIN** 11](#_Toc119765802)

[**LEFT JOIN** 12](#_Toc119765803)

[**RIGHT JOIN** 12](#_Toc119765804)

[**SQL JOINS using WHERE or ON** 13](#_Toc119765805)

[**SQL FULL OUTER JOIN** 14](#_Toc119765806)

[**SQL UNION** 14](#_Toc119765807)

[**COMPARISON WITH JOINS** 15](#_Toc119765808)

[**JOIN ON MULTIPLE KEYS** 16](#_Toc119765809)

[**SELF JOINS** 16](#_Toc119765810)

# **Aggregation functions**

Text

Description automatically generated

Figure 1. Aggregation functions.

# SQL Count

* Function that counts the number of rows in a particular column. Ie.

SELECT COUNT(\*)

* FROM tutorial.aapl\_historical\_stock\_price
* Using *COUNT(1)* has the same effect than *COUNT(\*)*. Which one to use, it’s just a matter of preferences.
* When we use *COUNT* over a specific column, the result it’s the number of *NOT NULL* rows for that column, no matter the type of such column Ie.

SELECT COUNT(high)

FROM tutorial.aapl\_historical\_stock\_price

* Practice problem: Write a query to count the number of non-null rows in the low column.

Select count(low)

From tutorial.aapl\_historical\_stock\_price

* To set a more readable name for the column, we just have to add the AS new\_colname, next to the clause COUNT(column). Ie.

SELECT COUNT(date) AS count\_of\_date

FROM tutorial.aapl\_historical\_stock\_price

* If the name we give to the column contains blank spaces, then we have to enclose the name between double quotes. Ie.

SELECT COUNT(date) AS "Count Of Date"

FROM tutorial.aapl\_historical\_stock\_price

* Note: This is the only place in which you’ll ever want to use double quotes in SQL. Single quotes for everything else.
* Problem: Write a query that determines counts of every single column. With these counts, can you tell which column has the most null values?

SELECT COUNT(year) AS year,

COUNT(month) AS month,

COUNT(open) AS open,

COUNT(high) AS high,

COUNT(low) AS low,

COUNT(close) AS close,

COUNT(volume) AS volume

FROM tutorial.aapl\_historical\_stock\_price

# **SQL SUM**

* SQL Function that totals the values in a given column.
* It can be used ONLY over numerical columns. Ie.

SELECT SUM(volume)

FROM tutorial.aapl\_historical\_stock\_price

* *AGGREGATORS ONLY AGGREGATE VERTICALLY*
* *Practice problem: Write a query to calculate the average opening price (hint: you will need to use both COUNT and SUM, as well as some simple arithmetic.).*

SELECT SUM(open)/COUNT(open) AS avg\_open\_price

FROM tutorial.aapl\_historical\_stock\_price

# **SQL MIN/MAX**

* Functions that return the lowest/highest value in a particular column.
* Can be user over numerical and non-numerical columns (as COUNT)
* Depending on the column type, MIN will return (MAX will do the opposite):
  + The lowest number.
  + The earliest date.
  + Closest non-numerical value to “A”.

SELECT MIN(volume) AS min\_volume,

MAX(volume) AS max\_volume

FROM tutorial.aapl\_historical\_stock\_price

* *Practice problem: What was Apple's lowest stock price (at the time of this data collection)?*

SELECT MIN(low)

FROM tutorial.aapl\_historical\_stock\_price

* *Practice problem: What was the highest single-day increase in Apple's share value?*

SELECT MAX(close - open)

FROM tutorial.aapl\_historical\_stock\_price

# **SQL AVERAGE**

* It returns the average of the selected group of values.
* It has limitations:
  + It can be used only on numerical columns.
  + It ignores null values completely.

SELECT AVG(high)

FROM tutorial.aapl\_historical\_stock\_price

WHERE high IS NOT NULL

* The above query produces the same result as the following query:

SELECT AVG(high)

FROM tutorial.aapl\_historical\_stock\_price

* For some cases is useful to treat null values as 0, by turning them into 0 values. (We’ll see it later on)
* *Practice problems: Write a query that calculates the average daily trade volume for Apple stock.*

SELECT AVG(volume) AS avg\_volume

FROM tutorial.aapl\_historical\_stock\_price

# **SQL GROUP BY**

* It allows us to separate data into groups unlike aggregate functions which aggregate across the entire table. Ie.

SELECT year,

COUNT(\*) AS count

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY year

* It’s possible to GROUP BY multiple columns, but then you have to separate them with commas (Just as in ORDER BY clause). Ie.

SELECT year,

month,

COUNT(\*) AS count

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY year, month

* *Practice problem: Calculate the total number of shares traded each month. Order your results chronologically.*

SELECT year, month, SUM(volume) AS volume\_sum

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY year, month

ORDER BY year, month

* Column names can be substituted by number on the GROUP BY clause. This generally recommended whenever you’re grouping many columns or if the text is excessively long. This supported by MODE, but it might not be supported on other platforms. Ie.

SELECT year, month, COUNT(\*) AS count

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY 1, 2

* When you GROUP BY multiple columns, you have to be aware of the fact that Aggregation functions are evaluated before LIMIT clause, so if you don’t GROUP BY any columns, you’ll get 1-row result.
* If you GROUP BY a column with enough unique values that it exceeds the LIMIT number, the aggregates will be calculated, and then some rows will simple be omitted from the result.
* *Practice problem: Write a query to calculate the average daily price change in Apple stock, grouped by year.*

SELECT year, AVG(close - open) AS avg\_daily\_change

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY 1

ORDER BY 1

* Practice problem: Write a query that calculates the lowest and highest prices that Apple stock achieved each month.

SELECT year, month, MIN(low) AS lowest\_price,

MAX(high) AS highest\_price

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY 1, 2

ORDER BY 1, 2

# SQL HAVING

* It’s the way of filtering aggregate columns (as where does with non-aggregate columns). i.e.

SELECT year, month, MAX(high) AS month\_high

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY year, month

HAVING MAX(high) > 400

ORDER BY year, month

* Note: Having is the “clean” way of filtering a query that has been aggregated, but this is also commonly done using a subqueries.
* *QUERY CLAUSE – EXECUTION ORDER:* SELECT 🡪 FROM 🡪 WHERE 🡪 GROUP BY 🡪 HAVING 🡪 ORDER BY

# **SQL CASE**

* Is the way of handling if/then logic. This clause is followed by at least one pair of WHEN and THEN statements.
* Every case statement must end with the END statement. The else statement is optional. Ie.

SELECT player\_name, year,

CASE WHEN year = 'SR' THEN 'yes'

ELSE NULL END AS is\_a\_senior

FROM benn.college\_football\_players

* The previous example goes as follow:
  + If the CASE statement is evaluated to TRUE, then “YES” gets printed in the column named is\_a\_senior.
  + If the CASE statement is evaluated to FALSE, then a NULL value is left in the column named is\_a\_senior
* To avoid the NULL value and set a value of 0. We’d have to do:

SELECT player\_name, year,

CASE WHEN year = 'SR' THEN 'yes'

ELSE 'no' END AS is\_a\_senior

FROM benn.college\_football\_players

* *Practice problem: Write a query that includes a column that is flagged "yes" when a player is from California and sort the results with those players first.*

SELECT player\_name, state,

CASE WHEN state = 'CA' THEN 'yes'

ELSE NULL END AS from\_california

FROM benn.college\_football\_players

ORDER BY 3

* It’S also possible to add multiple conditions to the CASE statement. Have in mind that it’s a good practice to avoid the overlapping on the conditions (with operators >= and <=). Ie.

SELECT player\_name, weight,

CASE WHEN weight > 250 THEN 'over 250'

WHEN weight > 200 AND weight <= 250 THEN '201-250'

WHEN weight > 175 AND weight <= 200 THEN '176-200'

ELSE '175 or under' END AS weight\_group

FROM benn.college\_football\_players

* *Practice problems: Write a query that includes players' names and a column that classifies them into four categories based on height. Keep in mind that the answer we provide is only one of many possible answers, since you could divide players' heights in many ways.*

SELECT player\_name, height,

CASE WHEN height > 74 THEN 'over 74'

WHEN height > 72 AND height <= 74 THEN '73-74'

WHEN height > 70 AND height <= 72 THEN '71-72'

ELSE 'under 70' END AS height\_group

FROM benn.college\_football\_players

* It’s also possible to string together multiple conditional statement with AND and OR the same way we’d do it with a WHERE clause.

SELECT player\_name,

CASE WHEN year = 'FR' AND position = 'WR' THEN 'frosh\_wr'

ELSE NULL END AS sample\_case\_statement

FROM benn.college\_football\_players

* GOLD RULES OF CASE STATEMENT:
  + Always goes in the SELECT clause.
  + Must include WHEN, THEN and END. ELSE is optional.
  + Conditional operators are accepted (like in WHERE clause) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
  + Multiple WHEN statements are accepted, as well as an ELSE statement to deal with any unaddressed conditions
* *Practice problem: Write a query that selects all columns from benn.college\_football\_players and adds an additional column that displays the player's name if that player is a junior or senior.*

SELECT \*,

CASE WHEN year IN ('JR', 'SR') THEN player\_name

ELSE NULL END AS upperclass\_player\_name

FROM benn.college\_football\_players

## **CASE WITH AGGREGATE FUNCTIONS**

* It becomes a bit more complicated and way more useful when is paired with aggregate functions. Ie.

SELECT

CASE WHEN year = 'FR' THEN 'FR'

ELSE 'Not FR'

END AS year\_group,

COUNT(1) AS count

FROM benn.college\_football\_players

GROUP BY CASE WHEN year = 'FR' THEN 'FR'

ELSE 'Not FR' END

* Counting multiples conditions in one query is one of the benefits of using aggregate functions with CASE statement. Also is important to have in mind that we can use either numbers or the column’s alias in the GROUP BY clause. Ie.

SELECT CASE WHEN year = 'FR' THEN 'FR'

WHEN year = 'SO' THEN 'SO'

WHEN year = 'JR' THEN 'JR'

WHEN year = 'SR' THEN 'SR'

ELSE 'No Year Data' END AS year\_group,

COUNT(1|year\_group) AS count

FROM benn.college\_football\_players

GROUP BY 1

* *Practice problem: Write a query that counts the number of 300lb+ players for each of the following regions: West Coast (CA, OR, WA), Texas, and Other (everywhere else).*

SELECT CASE WHEN state IN ('CA', 'OR', 'WA') THEN 'West Coast'

WHEN state = 'TX' THEN 'Texas'

ELSE 'Other' END AS arbitrary\_regional\_designation,

COUNT(1) AS players

FROM benn.college\_football\_players

WHERE weight >= 300

GROUP BY 1

* *Practice problem: Write a query that calculates the combined weight of all underclass players (FR/SO) in California as well as the combined weight of all upperclass players (JR/SR) in California.*

SELECT CASE WHEN year IN ('FR', 'SO') THEN 'underclass'

WHEN year IN ('JR', 'SR') THEN 'upperclass'

ELSE NULL END AS class\_group,

SUM(weight) AS combined\_player\_weight

FROM benn.college\_football\_players

WHERE state = 'CA'

GROUP BY 1

## **USING CASE INSIDE OF AGGREGATION FUNCTIONS**

* This is known as pivoting and consists of displaying information horizontally. Ie. The following query is showing information vertically.

SELECT CASE WHEN year = 'FR' THEN 'FR'

WHEN year = 'SO' THEN 'SO'

WHEN year = 'JR' THEN 'JR'

WHEN year = 'SR' THEN 'SR'

ELSE 'No Year Data' END AS year\_group,

COUNT(1) AS count

FROM benn.college\_football\_players

GROUP BY 1

A horizontal version of this query would be.  
SELECT COUNT(CASE WHEN year = 'FR' THEN 1 ELSE NULL END) AS fr\_count,

COUNT(CASE WHEN year = 'SO' THEN 1 ELSE NULL END) AS so\_count,

COUNT(CASE WHEN year = 'JR' THEN 1 ELSE NULL END) AS jr\_count,

COUNT(CASE WHEN year = 'SR' THEN 1 ELSE NULL END) AS sr\_count

FROM benn.college\_football\_players

* *Practice problem: Write a query that displays the number of players in each state, with FR, SO, JR, and SR players in separate columns and another column for the total number of players. Order results such that states with the most players come first.*

SELECT state,

COUNT(CASE WHEN year = 'FR' THEN 1 ELSE NULL END) AS fr\_count,

COUNT(CASE WHEN year = 'SO' THEN 1 ELSE NULL END) AS so\_count,

COUNT(CASE WHEN year = 'JR' THEN 1 ELSE NULL END) AS jr\_count,

COUNT(CASE WHEN year = 'SR' THEN 1 ELSE NULL END) AS sr\_count,

COUNT(1) AS total\_players

FROM benn.college\_football\_players

GROUP BY state

ORDER BY total\_players DESC

* *Practice problem: Write a query that shows the number of players at schools with names that start with A through M, and the number at schools with names starting with N - Z.*

SELECT CASE WHEN school\_name < 'n' THEN 'A-M'

WHEN school\_name >= 'n' THEN 'N-Z'

ELSE NULL END AS school\_name\_group,

COUNT(1) AS players

FROM benn.college\_football\_players

GROUP BY 1

# **SQL DISTINCT**

* Use it to look at unique values in a particular column. Ie.

SELECT DISTINCT month

FROM tutorial.aapl\_historical\_stock\_price

* When is used over two or more columns, the result is all of the unique pairs of those columns. Ie.

SELECT DISTINCT year, month

FROM tutorial.aapl\_historical\_stock\_price

* NOTE: It’s needed just one time, so don’t include it for every column name.
* *Practice problem: Write a query that returns the unique values in the year column, in chronological order.*

SELECT DISTINCT year

FROM tutorial.aapl\_historical\_stock\_price

* ORDER BY year
* Can be useful due to identifying all of the unique values, can lead to identify how you might want to group or filter the data.

## **DISTINCT IN AGGREGATION**

* Most of the times is combined with the COUNT function. Ie.

SELECT COUNT(DISTINCT month) AS unique\_months

FROM tutorial.aapl\_historical\_stock\_price

* As you might have noticed, DISTINCT goes inside of the aggregation function. So, you also could SUM or AVG DISTINCT values in a column, but there are fewer practical applications for them.
* You don’t want to use DISTINCT with MIN/MAX due to the result will be the same, with the downside that the query will be slower.
* It’s important to have in mind that DISTINCT can make your queries a bit slower, particularly in aggregations.
* *Practice problem: Write a query that counts the number of unique values in the month column for each year.*

SELECT year, COUNT(DISTINCT month) AS months\_count

FROM tutorial.aapl\_historical\_stock\_price

GROUP BY year

ORDER BY year

* *Practice problem: Write a query that separately counts the number of unique values in the month column and the number of unique values in the `year` column.*

SELECT COUNT(DISTINCT year) AS years\_count,

COUNT(DISTINCT month) AS months\_count

FROM tutorial.aapl\_historical\_stock\_price

# **SQL JOINS**

* JOIN is the statement that allows to make queries about the information stored in more than one table. JOIN statement is followed by a table name and the ON statement, which is followed by a couple of names separated by an equals signs (name\_field\_table1 = name\_field\_table2)
* The ON statement indicates how the tables relate to each other.
* These relationships are called mapping sometimes.
* The columns that are mapped to another in a different table, are called foreign keys or join keys. They are written as a conditional statement --> ON teams.school\_name = players.school\_name.
* In plain English, what JOIN does is: Join all rows from the players table on to rows in the teams table for which the school\_name field in the players table is equal to the school\_name field in the teams table.
* During the join operation, SQL looks up the school\_name (following the previous example) in the school\_name field of the teams table. If it finds a match, SQL takes all of the columns from the teams table and joins them to the columns of the players table. The new result will have all the columns of both tables. This process is repeated for every row of table which is after the FROM statement. This is an example of a full SQL statement with a join and aggregation functions.

SELECT teams.conference AS conference,

AVG(players.weight) AS average\_weight

FROM benn.college\_football\_players players

JOIN benn.college\_football\_teams teams

ON teams.school\_name = players.school\_name

GROUP BY teams.conference

ORDER BY AVG(players.weight) DESC

* *Practice problems: Write a query that selects the school name, player name, position, and weight for every player in Georgia, ordered by weight (heaviest to lightest). Be sure to make an alias for the table, and to reference all column names in relation to the alias.*

SELECT players.school\_name,

players.player\_name,

players.position,

players.weight

FROM benn.college\_football\_players players

WHERE players.state = 'GA'

ORDER BY players.weight DESC

# **INNER JOIN**

* It can be written as JOIN or INNER JOIN.
* Eliminates rows from both table that don’t satisfy the join condition set in the ON statement.
* In mathematical terms, is the intersection of the two tables (view Fig 2)

Diagram, venn diagram

Description automatically generated

Figure 2. Inner join.

## **JOINING TABLES WITH IDENTICAL COLUMN NAMES**

* As the result of a join only can support one column with a given name. When you include 2 columns whose names are the same, the result will show the exact same result set for both columns, even if the two columns should contain different data.
* They way of avoiding problems for these cases, is naming the columns individually. Ie.

SELECT players.school\_name AS players\_school\_name,

teams.school\_name AS teams\_school\_name

FROM benn.college\_football\_players players

JOIN benn.college\_football\_teams teams

ON teams.school\_name = players.school\_name

* *Practice problems: Write a query that displays player names, school names and conferences for schools in the "FBS (Division I-A Teams)" division.*

SELECT players.player\_name,

players.school\_name,

teams.conference

FROM benn.college\_football\_players players

JOIN benn.college\_football\_teams teams

ON teams.school\_name = players.school\_name

WHERE teams.division = 'FBS (Division I-A Teams)'

# **SQL OUTER JOIN**

* Joins that return matched values and unmatched values from either or both tables. These are a few types of outer joins: There are a few types of outer joins:
  + *Left Join:*  *Return only unmatched rows from the left table*, as well as matched rows in both tables.
  + *Right Join:*  *Return only unmatched rows from the right table*, as well as matched rows in both tables.
  + *Full outer Join: Return unmatched rows from both tables*, as well as matched rows in both tables.
  + *These joins are also referred as: OUTER LEFT JOIN, OUTER RIGHT JOIN and OUTER JOIN.*
* In the Figure 3, it can be seen how every kind of JOIN looks like.

Graphical user interface

Description automatically generated

Figure 3. Visual Join.

## **LEFT JOIN**

* *Practice problem: Write a query that performs an inner join between the tutorial.crunchbase\_acquisitions table and the tutorial.crunchbase\_companies table, but instead of listing individual rows, count the number of non-null rows in each table.*

SELECT COUNT(companies.permalink) AS companies\_rowcount,

COUNT(acquisitions.company\_permalink) AS acquisitions\_rowcount

FROM tutorial.crunchbase\_companies companies

JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

* *Practice problem: Write a query that performs an inner join between the tutorial.crunchbase\_acquisitions Modify the query above to be a LEFT JOIN. Note the difference in results.*

SELECT COUNT(companies.permalink) AS companies\_rowcount,

COUNT(acquisitions.company\_permalink) AS acquisitions\_rowcount

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

* *Practice problem: Count the number of unique companies (don't double-count companies) and unique acquired companies by state. Do not include results for which there is no state data, and order by the number of acquired companies from highest to lowest.*

SELECT companies.state\_code,

COUNT(DISTINCT companies.permalink) AS unique\_companies,

COUNT(DISTINCT acquisitions.company\_permalink) AS unique\_companies\_acquired

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

WHERE companies.state\_code IS NOT NULL

GROUP BY 1

ORDER BY 3 DESC

## **RIGHT JOIN**

* These are rarely used due to the same results can be reached with left joins.
* There’s actually a sort of convention that encourage people to use left join instead right join, but actually nothing forbids you to use right join whenever you fancy it.
* *Practice problem: Rewrite the previous practice query in which you counted total and acquired companies by state, but with a RIGHT JOIN instead of a LEFT JOIN. The goal is to produce the exact same results.*

SELECT companies.state\_code,

COUNT(DISTINCT companies.permalink) AS unique\_companies,

COUNT(DISTINCT acquisitions.company\_permalink) AS acquired\_companies

FROM tutorial.crunchbase\_acquisitions acquisitions

RIGHT JOIN tutorial.crunchbase\_companies companies

ON companies.permalink = acquisitions.company\_permalink

WHERE companies.state\_code IS NOT NULL

GROUP BY 1

ORDER BY 3 DESC

# **SQL JOINS using WHERE or ON**

* Filtering is usually carried out in the WHERE clause, although doing it before joining two tables is possible as well. i.e.

SELECT companies.permalink AS companies\_permalink,

companies.name AS companies\_name,

acquisitions.company\_permalink AS acquisitions\_permalink,

acquisitions.acquired\_at AS acquired\_date

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

AND acquisitions.company\_permalink != '/company/1000memories'

ORDER BY 1

* On the above example, the AND statement is evaluated before than the join happens. This is a sort of WHERE clause that only applies to one of the tables.
* When the filtering is carried out on the WHERE clause, this one is produced after joining the tables. Filtering in the where clause can also filter null values. Ie.

SELECT companies.permalink AS companies\_permalink,

companies.name AS companies\_name,

acquisitions.company\_permalink AS acquisitions\_permalink,

acquisitions.acquired\_at AS acquired\_date

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_acquisitions acquisitions

ON companies.permalink = acquisitions.company\_permalink

WHERE acquisitions.company\_permalink != '/company/1000memories'

OR acquisitions.company\_permalink IS NULL

ORDER BY 1

* *Practice problem: Write a query that shows a company's name, "status" (found in the Companies table), and the number of unique investors in that company. Order by the number of investors from most to fewest. Limit to only companies in the state of New York.*

SELECT companies.name AS company\_name,

companies.status,

COUNT(DISTINCT investments.investor\_name) AS unqiue\_investors

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_investments investments

ON companies.permalink = investments.company\_permalink

WHERE companies.state\_code = 'NY'

GROUP BY 1,2

ORDER BY 3 DESC

* *Practice problem: Write a query that lists investors based on the number of companies in which they are invested. Include a row for companies with no investor, and order from most companies to least.*

SELECT CASE WHEN investments.investor\_name IS NULL THEN 'No Investors'

ELSE investments.investor\_name END AS investor,

COUNT(DISTINCT companies.permalink) AS companies\_invested\_in

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_investments investments

ON companies.permalink = investments.company\_permalink

GROUP BY 1

ORDER BY 2 DESC

# **SQL FULL OUTER JOIN**

* It’S quite unlikely to be used.
* Is commonly used in conjunction with aggregation functions to understand the amount of overlap between two tables
* *Practice problem: Write a query that joins tutorial.crunchbase\_companies and tutorial.crunchbase\_investments\_part1 using a FULL JOIN. Count up the number of rows that are matched/unmatched as in the example above.*

# **SQL UNION**

* It allows you to stack one dataset on top of the other.
* It allows you to write two different SELECT statements, giving a table result with the columns of both SELECTS (First SELECT results first and then, the second).
* Basically, what happens in a UNION is that the second datasets are appended and Only include distinct values. (if a row was already in the first dataset, then such row is omitted)
* If you’d like to append all of the values from the second table, the use UNION ALL.
* Strict rules when appending data:
  + Both tables must have te same number of columns.
  + The columns must have the same data types in the same order as the first table.
* The column names don’t have to be the same, they usually are though.
* It’s possible to treat the two datasets differently. i.e., Filtering them differently using different WHERE clause.
* Example of union [ALL]

SELECT \*

FROM tutorial.crunchbase\_investments\_part1

UNION [ALL]

SELECT \*

FROM tutorial.crunchbase\_investments\_part2

* *Practice problem: Write a query that appends the two crunchbase\_investments datasets above (including duplicate values). Filter the first dataset to only companies with names that start with the letter "T", and filter the second to companies with names starting with "M" (both not case-sensitive). Only include the company\_permalink, company\_name, and investor\_name columns.*

SELECT company\_permalink,

company\_name,

investor\_name

FROM tutorial.crunchbase\_investments\_part1

WHERE company\_name ILIKE 'T%'

UNION ALL

SELECT company\_permalink,

company\_name,

investor\_name

FROM tutorial.crunchbase\_investments\_part2

WHERE company\_name ILIKE 'M%'

# **COMPARISON WITH JOINS**

* Comparison operators can be used either in the ON statement or in the WHERE clause of a join. Ie.

SELECT companies.permalink,

companies.name,

companies.status,

COUNT(investments.investor\_permalink) AS investors

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_investments\_part1 investments

ON companies.permalink = investments.company\_permalink

AND investments.funded\_year > companies.founded\_year + 5

GROUP BY 1,2, 3

* It’S important to stand out that the effect caused by the operator will be different depending on if this one has been used in the ON statement or in the WHERE clause.

SELECT companies.permalink,

companies.name,

companies.status,

COUNT(investments.investor\_permalink) AS investors

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_investments\_part1 investments

ON companies.permalink = investments.company\_permalink

WHERE investments.funded\_year > companies.founded\_year + 5

GROUP BY 1,2, 3

# **JOIN ON MULTIPLE KEYS**

* One the reasons you want to join on multiple foreign keys, has to do with accuracy.
* The second reason has to do with performance, due to indexes make your queries faster when you join on multiple fields. It’s worth noting that this will have relatively little effect on small dataset. Ie.

SELECT companies.permalink,

companies.name,

investments.company\_name,

investments.company\_permalink

FROM tutorial.crunchbase\_companies companies

LEFT JOIN tutorial.crunchbase\_investments\_part1 investments

ON companies.permalink = investments.company\_permalink

AND companies.name = investments.company\_name

# **SELF JOINS**

* It can be useful sometimes join a table to itself
* Same table can easily be reference multiples times using different aliases.

SELECT DISTINCT japan\_investments.company\_name,

japan\_investments.company\_permalink

FROM tutorial.crunchbase\_investments\_part1 japan\_investments

JOIN tutorial.crunchbase\_investments\_part1 gb\_investments

ON japan\_investments.company\_name = gb\_investments.company\_name

AND gb\_investments.investor\_country\_code = 'GBR'

AND gb\_investments.funded\_at > japan\_investments.funded\_at

WHERE japan\_investments.investor\_country\_code = 'JPN'

ORDER BY 1