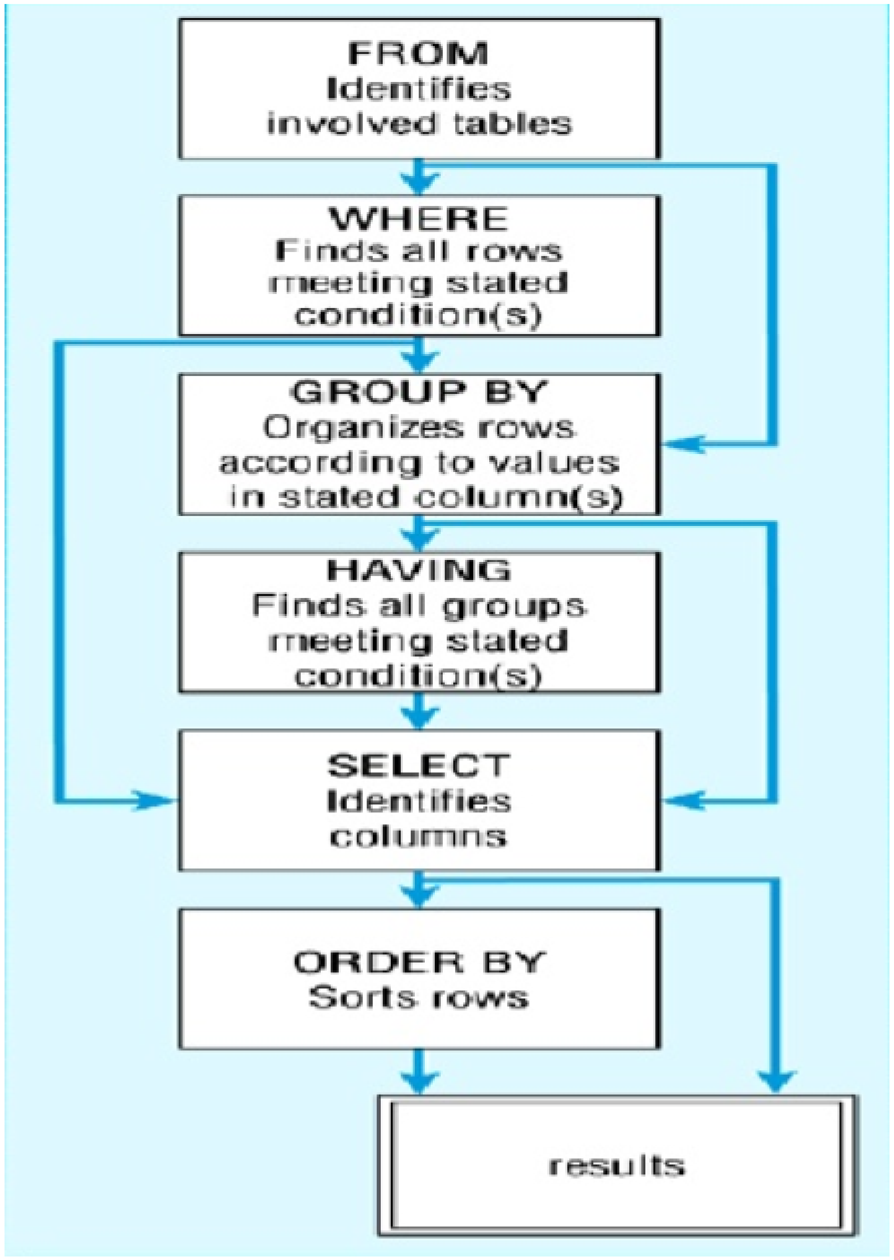
Logical Query Processing Order

- Being familiar with the logical query processing order can be of great help in understanding and writing SQL queries. It helps answer questions such as: why WHERE statement cannot be used with aggregate functions? Why GROUP BY must be after WHERE, while HAVING must be after GROUP by, etc. A skeletal diagram for logical query processing order is shown below, after which there are brief description of each step.


FROM
Identifies involved tables
WHERE
Finds all rows meeting stated condition(s)
GROUP BY
Organizes rows according to values in stated column(s)
HAVING
Finds all groups meeting stated condition(s)
SELECT
Identifies columns
ORDER BY
Sorts rows
results

- FROM: followed by data source (table). The data source can be, single table, multiple tables, table from subquery, from joining, etc.
- WHERE: Row-by-row filtering with specified condition. When WHERE is doing row filtering, it has no idea on the aggregated quantity of the table. Thus WHERE cannot be used with aggregate functions.

- GROUP BY: Arrange (WHERE) filtered rows into groups. So it must be after where statement if there is one. Aggregate functions are often used with GROUP BY. These functions are applied to each group, similar to their usage on an entire table.
- HAVING: Unlike the row filtering function of WHERE, HAVING filters groups. Thus HAVING must be after GROUP BY. HAVING can be used with aggregate functions, which are applied to each individual group.
- SELECT: It is executed almost in the end, although it is written in the first place.
- ORDER BY: If ORDER BY is not in the last place, then previously ordered rows might be affected by other actions after ORDER BY.

More on group by

- 'GROUP BY A': In a group indexed by A, there might be many records corresponding to the same A. Thus we usually apply aggregate action on each subgroup.
- 'GROUP BY A, B,...': Sometimes, repeated records occur even for the same A, B,... In other words, the selected group indexed by A, B,... might still correspond to many records.
- Many problems like printing out 'column1, aggregate()' can be solved by a simple query with GROUP BY. Sometimes even printing out 'column1, column2, aggregate()' can also be solved by a simple query with GROUP BY. These correspond to the two situations mentioned above. However, printing out 'column1, column2 (or even more columns), aggregate() most probably cannot be solved by a single query with GROUP BY. See the example in the advanced subquery in Data Camp course, where we have two ways to handle this: Join or subquery.
- (1) Whenever a problem involves 'every, per column_name…', then it usually needs GROUP BY. (2) Build a Query to Determine the Percentage of Population by Gender and State. Note the key word 'by'… this is also related to group by. However, 'By Gender and State' does not mean we have to use "select gender, state, aggregateFunc().... group by gender, state. The gender might be incorporated into the aggregate function. The key is to follow the query logical order. Here, grouping first, and then operate (select) on the groups.
- When GROUPING BY A, B, C, we can SELECT ONLY SUBSET of (A, B, C), or aggregate for **ANY** column. In other words, we cannot group by columns A, B, C but select D,E,... This is true in SQL. However, in pandas, we **don't have such a restriction in Pandas**. See details in the relevant notes of Pandas.

Basic query examples

Better identify each problem with the road map outlined

- select title, release_year from films where release_year BETWEEN 1990 and 2000
- select title, language from films where language IN ('English', 'Spanish','French');
- select name from people where name LIKE 'B%'; name start with B, regular exp.

- select name from people where name LIKE `'_r%'` ; second letter is r.
- select name from people where name NOT LIKE 'A%'; name not start from A. In SQL, != sometimes OK, but standard seems to be <>. Also we normally use = but not ==.

Get the amount grossed by the best performing film between 2000 and 2012, inclusive:

- select MAX(gross) from films where release_year BETWEEN 2000 AND 2012
- select `45/10*100.0` will give a 400, which is wrong. select `45*100.0/10` will give the correct number 450. So when you are dividing make sure at least one of your numbers has a decimal place. This rule apply in some SQL servers.

Get the percentage of people who are no longer alive. Alias the result as percentage_dead. Remember to use 100.0 and not 100!

- select `count(deathdate)*100.0/count(*)` as percentage_dead from people;
  Another way:
- select `count(deathdate) / count(*) * 100.0` as percentage_dead from people;
  The above is completely wrong, even we use 100.0. See reasons earlier explained.
- select IMDB_score,ID from reviews order by IMDB_score DESC.
- select birthdate,name from people order by birthdate,name; order by two columns, meaning order by first column first, and then order by second.

Get the release year and count of films released in each year.

- select release_year, `count(*)` from films group by release_year.

Get the release year, country, and highest budget spent making a film for each year, for each country. Sort your results by release year and country. Do a practice before checking the following answer.

- select release_year, country, max(budget) from films group by release_year, country order by release_year, country;

A good example: In how many different years were more than 200 movies released?

Method 1:

- select count `(*)` from
  (
  select release_year
  from films
  group by release_year
  Having count(title) > 200
) as a;

Note without the 'as a' or just 'a' then it will have problems. **Also note 'having' to subgroup is just like where to single row. Using 'having' properly can save further sub-querying**.

Method 2:

- select release_year
  into #temp
  ...
  select count(*) from #temp.

Another example. Get the country, average budget, and average gross take of countries that have made more than 10 films. Order the result by country name, and limit the number of results displayed to 5. You should alias the averages as avg_budget and avg_gross respectively.

- select country, AVG(budget) as avg_budget, AVG(gross) as avg_gross
- from films
- group by country
- Having count(title) > 10
- order by country
- limit 5
  The 'having count(title) > 10' should be same as 'having `count(*)'` here?

Comments: Can the above sub-queries be transformed into joins easily?