Study Guide: Data Retrieval with SQL

Afshine Amidi and Shervine Amidi

August 9, 2020

General concepts

❒ **Structured Query Language** – Structured Query Language, abbreviated as SQL, is a language that is used to query data from databases and is largely used in the industry.

❒ **Query structure** – Queries are usually structured as follows:

SQL

```
-- Select fields                    mandatory
SELECT
    field_1,
    field_2,
      ...  ,
    field_n

-- Source of data                   mandatory
FROM table t

-- Gather info from other sources   optional
JOIN other_table ot
  ON (t.key = ot.key)

-- Conditions                       optional
WHERE some_condition(s)

-- Aggregating                      optional
GROUP BY column_group_list

-- Sorting values                   optional
ORDER BY column_order_list

-- Restricting aggregated values    optional
HAVING some_condition(s)

-- Limiting number of rows          optional
LIMIT some_value
```

Remark: the `SELECT DISTINCT` command can be used to ensure not having duplicate rows.

❒ **Condition** – A condition is of the following format:

SQL

```
some_field some_operator some_field
```

where `some_operator` can be among the following common operations:

Category	Operator	Command
General	Equality / non-equality	`= / !=, <>`
	Inequalities	`>=, >, <, <=`
	Belonging	`IN (val_1, ..., val_n)`
	And / or	`AND / OR`
	Check for missing value	`IS NULL`
	Between bounds	`BETWEEN val_1 AND val_2`
Strings	Pattern matching	`LIKE '%val%'`

❒ **Joins** – Two tables `table_1` and `table_2` can be joined in the following way:

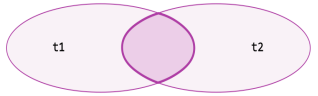
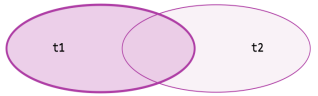
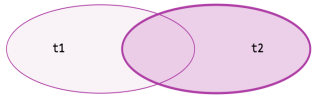
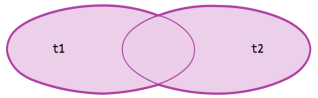
SQL

```
...

FROM table_1 t1
type_of_join table_2 t2
  ON (t2.key = t1.key)

...
```

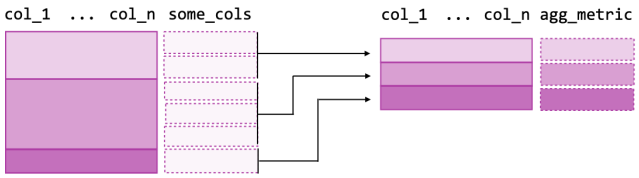
where the different `type_of_join` commands are summarized in the table below:

Type of join	Illustration
INNER JOIN	
LEFT JOIN	
RIGHT JOIN	
FULL JOIN	

Remark: joining every row of table 1 with every row of table 2 can be done with the `CROSS JOIN` command, and is commonly known as the cartesian product.

Aggregations

❒ **Grouping data** – Aggregate metrics are computed on grouped data in the following way:



The SQL command is as follows:

SQL

```
SELECT
    field_1,
    agg_function(field_2)
FROM table
GROUP BY field_1
```

❒ **Grouping sets** – The `GROUPING SETS` command is useful when there is a need to compute aggregations across different dimensions at a time. Below is an example of how all aggregations across two dimensions are computed:

SQL

```
SELECT
    field_1,
    field_2,
    agg_function(field_3)
FROM table
GROUP BY (
  GROUPING SETS
    (field_1),
    (field_2),
    (field_1, field_2)
)
```

❒ **Aggregation functions** – The table below summarizes the main aggregate functions that can be used in an aggregation query:

Category	Operation	Command
Values	Mean	`AVG(col)`
	Percentile	`PERCENTILE_APPROX(col, p)`
	Sum / # of instances	`SUM(col)` / `COUNT(col)`
	Max / min	`MAX(col)` / `MIN(col)`
	Variance / standard deviation	`VAR(col)` / `STDEV(col)`
Arrays	Concatenate into array	`collect_list(col)`

Remark: the median can be computed using the `PERCENTILE_APPROX` function with `p` equal to `0.5`.

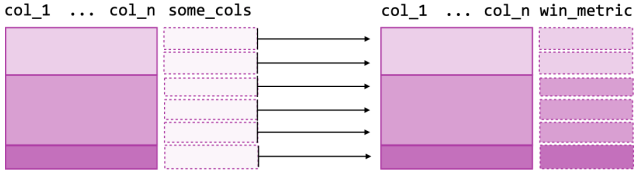
❒ **Filtering** – The table below highlights the differences between the `WHERE` and `HAVING` commands:

`WHERE`	`HAVING`
- Filter condition applies to individual rows - Statement placed right after `FROM`	- Filter condition applies to aggregates - Statement placed right after `GROUP BY`

Remark: if `WHERE` and `HAVING` are both in the same query, `WHERE` will be executed first.

Window functions

❒ **Definition** – A window function computes a metric over groups and has the following structure:



The SQL command is as follows:

SQL

```
some_window_function() OVER(PARTITION BY some_field ORDER BY another_field)
```

Remark: window functions are only allowed in the `SELECT` clause.

❒ **Row numbering** – Ranks each row across specified groups, ordered by a specific field. The table below summarizes the main commands:

Command	Description	Example
`ROW_NUMBER()`	Ties are given different ranks	1, 2, 3, 4
`RANK()`	Ties are given same rank and skip numbers	1, 2, 2, 4
`DENSE_RANK()`	Ties are given same rank and don't skip numbers	1, 2, 2, 3

❒ **Values** – The following window functions allow to keep track of specific types of values with respect to the partition:

Command	Description
`FIRST_VALUE(col)`	Takes the first value of the column
`LAST_VALUE(col)`	Takes the last value of the column
`LAG(col, n)`	Takes the n^{th} previous value of the column
`LEAD(col, n)`	Takes the n^{th} following value of the column
`NTH_VALUE(col, n)`	Takes the n^{th} value of the column

Advanced functions

❒ **SQL tips** – In order to keep the query in a clear and concise format, the following tricks are often done:

Operation	Command	Description
Renaming columns	`SELECT operation_on_column AS col_name`	New column names shown in query results
Abbreviating tables	`FROM table_1 t1`	Abbreviation used within query for simplicity in notations
Simplifying group by	`GROUP BY col_number_list`	Specify column position in `SELECT` clause instead of whole column names
Limiting results	`LIMIT n`	Display only `n` rows

❒ **Sorting values** – The query results can be sorted along a given set of columns using the following command:

SQL
```
... [query] ...
ORDER BY col_list
```

Remark: by default, the command sorts in ascending order. If we want to sort it in descending order, the `DESC` command needs to be used after the column.

❒ **Column types** – In order to ensure that a column or value is of one specific data type, the following command is used:

SQL
```
CAST(some_column_or_value AS data_type)
```

where `data_type` is one of the following:

Data type	Description	Example
`INT`	Integer	`2`
`DOUBLE`	Numerical value	`2.0`
`STRING` / `VARCHAR`	String	
`DATE`	Date	`'2020-01-01'`
`TIMESTAMP`	Timestamp	`'2020-01-01 00:00:00.000'`

Remark: if the column contains data of different types, the `TRY_CAST()` command will convert unknown types to `NULL` instead of throwing an error.

❒ **Column manipulation** – The main functions used to manipulate columns are described in the table below:

Category	Operation	Command
General	Take first non-`NULL` value	`COALESCE(col_1, col_2, ..., col_n)`
	Create a new column combining existing ones	`CONCAT(col_1, ..., col_n)`
Value	Round value to `n` decimals	`ROUND(col, n)`
String	Converts string column to lower / upper case	`LOWER(col)` / `UPPER(col)`
	Replace occurrences of `old` in col to `new`	`REPLACE(col, old, new)`
	Take the substring of col, with a given `start` and `length`	`SUBSTR(col, start, length)`
	Remove spaces from the left / right / both sides	`LTRIM(col)` / `RTRIM(col)` / `TRIM(col)`
Date	Truncate at a given granularity (`year`, `month`, `week`)	`DATE_TRUNC(time_dimension, field_date)`
	Transform date	`DATE_ADD(field_date, number_of_days)`

❒ **Conditional field** – – A column can take different values with respect to a particular set of conditions with the `CASE WHEN` command as follows:

SQL
```
CASE WHEN some_condition THEN some_value
                ...
     WHEN some_other_condition THEN some_other_value
     ELSE some_other_value_n END
```

❒ **Combining results** – The table below summarizes the main ways to combine results in queries:

Category	Command	Remarks
Union	`UNION`	Guarantees distinct rows
	`UNION ALL`	Potential newly-formed duplicates are kept
Intersection	`INTERSECT`	Keeps observations that are in all selected queries

❒ **Common table expression** – A common way of handling complex queries is to have temporary result sets coming from intermediary queries, which are called common table expressions (abbreviated CTE), that increase the readability of the overall query. It is done thanks to the `WITH ... AS ...` command as follows:

SQL
```
WITH cte_1 AS (
SELECT ...
),
```

```
...

cte_n AS (
SELECT ...
)

SELECT ...
FROM ...
```

Table manipulation

❒ **Table creation** – The creation of a table is done as follows:

SQL

```
CREATE [table_type] TABLE [creation_type] table_name(
  field_1 data_type_1,
        ...       ,
  field_n data_type_n
)
[options];
```

where `[table_type]`, `[creation_type]` and `[options]` are one of the following:

Category	Command	Description
Table type	Blank	Default table
	`EXTERNAL TABLE`	External table
Creation type	Blank	Creates table and overwrites current one if it exists
	`IF NOT EXISTS`	Only creates table if it does not exist
Options	`location 'path_to_hdfs_folder'`	Populate table with data from hdfs folder
	`stored as data_format`	Stores the table in a specific data format, e.g. parquet, orc or avro

❒ **Data insertion** – New data can either append or overwrite already existing data in a given table as follows:

SQL

```
WITH ...                          -- optional

INSERT [insert_type] table_name   -- mandatory

SELECT ...;                       -- mandatory
```

where `[insert_type]` is among the following:

Command	Description
`OVERWRITE`	Overwrites existing data
`INTO`	Appends to existing data

❒ **Dropping table** – Tables are dropped in the following way:

SQL

```
DROP TABLE table_name;
```

❒ **View** – Instead of using a complicated query, the latter can be saved as a view which can then be used to get the data. A view is created with the following command:

SQL

```
CREATE VIEW view_name AS complicated_query;
```

Remark: a view does not create any physical table and is instead seen as a shortcut.