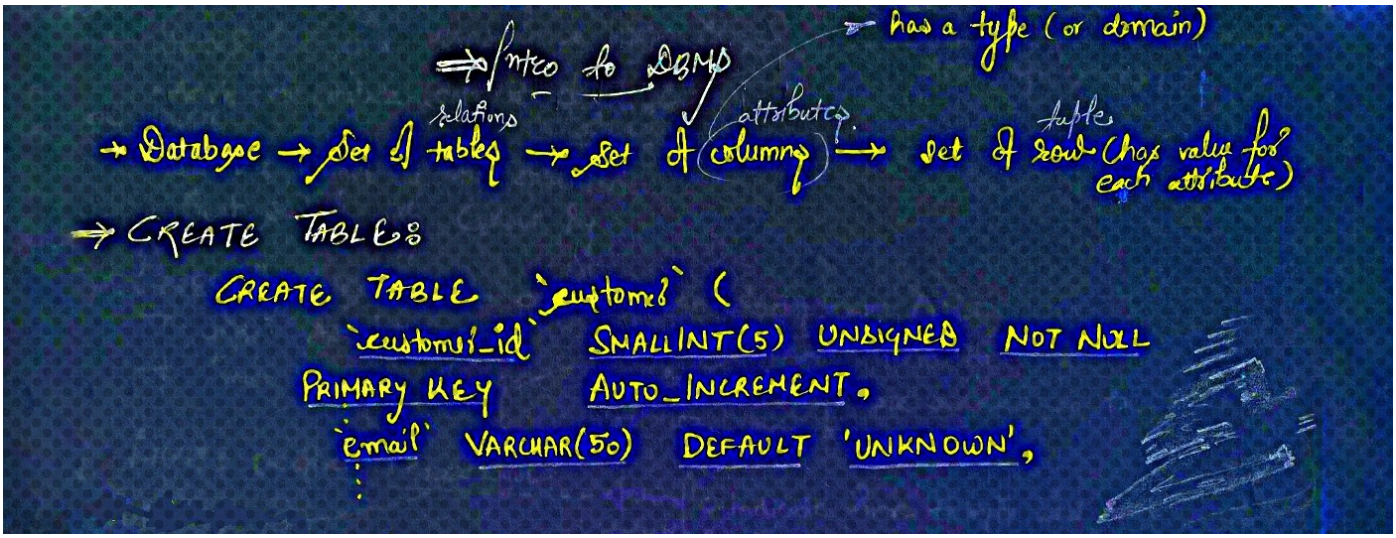
Published in Towards Data Science

Eklavya Saxena Follow

Feb 27, 2020 · 8 min read · Listen

SQL — Practical Details Cheat Sheet for Data Analysis

The most neglected yet the most important of all languages



Data is considered more valuable than oil due to insight and knowledge that can be extracted from it — hence the importance of Data Analytics. And it all starts from querying the database, making SQL its core. There are plentiful of SQL tutorials and cheat sheets available. Therefore, I will refrain from discussing the basics and will focus more on the most important aspects or practical details that I have learnt from my experience.

1. SELECT statement syntax — Data Query Language

4. Sub-queries — A Query within Query

. . .

1. SELECT statement syntax — Data Query Language

In this section, SELECT statement's MySQL syntax is described with most commonly used clauses and their practical details.

```
SELECT [select_option]
{select_expression}
[
  FROM
    JOIN
    ON
    WHERE
    GROUP BY
    HAVING
    ORDER BY
    LIMIT
    INTO
]
```

WHERE — Aliases & Aggregate Functions CAN NOT be used

It is NOT permissible to refer to a **column alias** in a WHERE clause, because the column value might not yet be determined when the WHERE clause is executed (explained in next section).

It is NOT permissible to refer to an **aggregate function** in a WHERE clause, because WHERE clause doesn't have access to entire set but is applied to each row as it is presented to the db engine. Whereas **aggregate functions** (e.g. SUM()) work on sets of data.

WHERE clause requires at least one condition. And, if multiple conditions, connect them using AND or OR logic operators.

> ***Good Practice:***




HAVING — works similar to WHERE but use cautiously

An advantage of HAVING clause over WHERE clause is that it filters the aggregated data whereas the latter filters the base data. In other words, WHERE clause acts as a **pre-filter** whereas HAVING clause as a **post-filter**. Hence, **aggregate functions** can be used with HAVING clause.

NOTE: Aliases in HAVING clause: Surprisingly, query successfully executes for 'column aliases' and fails for 'aggregate function' aliases.
Best practice is to **refrain** from using **aliases** in HAVING clause also.

Do not use HAVING for items that should be in the WHERE clause, because WHERE is more **optimized**.

GROUP BY — should include aggregate functions

SELECT list should include a function (e.g. SUM(), COUNT()) that summarizes the data. WITH ROLLUP modifier can be used to include extra rows that represent higher-level (that is, super-aggregate) summary operations.

```
SELECT cat, COUNT(*) AS total ... GROUP BY category WITH ROLLUP;
```

ORDER BY and LIMIT

ORDER BY clause should include at least one column name or alias. If more than one column is included, separated by a comma, then the columns are ordered in the order they are mentioned.

Take advantage of **[offset,]** modifier (default value = 0) of LIMIT clause which indicates after which row to begin the row count. For e.g. `LIMIT 10, 5` starts the row count after 10th row, and returns 5 rows — 11th, 12th…, 15th.

INTO — Makes SELECT work as DML (Data Manipulation Language)

SELECT … INTO form is considered to be DML because it manipulates (i.e. modifies) the data. For e.g. you have an empty table `` `table` `` and want to insert data from table `` `backup` `` . Then the following query will act as DML:




only the **last** value returned by the WHERE clause will be assigned to the variable. For e.g. if WHERE clause in the below query returns (horror, comedy, romantic) then `romantic ` will be assigned to the variable:

```
SELECT @category := category FROM films WHERE rating > 3;
```

. . .

2. Logical Processing Order of the SELECT statement

This section is briefly curated from a blog "SQL Order of Operations — In Which Order MySQL Executes Queries?" written by Tomer Shay @ EverSQL.
I hope this will enable you to write optimized queries with few hit and trials.

NOTE: The actual physical execution of the statement is determined by the **query processor** and the order may vary from the table below. Although, it will give the same result as if the query ran in below(default) execution order.

Order	SQL Parts	Function
1	FROM and JOINs	determines the entire working set – data from all tables according to the JOINs ON clauses, also subqueries
2	WHERE clause	filters the data according to the conditions
3	GROUP BY clause	aggregates the data according to one/more columns - splitting data to different chunks or buckets, where each bucket has one key and a list of rows that match that key
4	HAVING clause	filters the aggregated data - used to filter out some buckets
5	WINDOW functions	just like grouping - performs calculation on a set of rows but each row will keep its own identity and won't be grouped into a bucket of other similar rows
6	SELECT clause	selects data after filtering & grouping - use column names, aggregations and subqueries inside it
7	DISTINCT keyword	discard rows with duplicate values from remaining rows left after the filtering and aggregations
8	UNION keyword	combines the result sets of two queries into one result set
9	ORDER BY clause	sorts the entire result set using columns, aliases, aggregate functions, even if not part of selected data. NOTE: using DISTINCT prevents sorting by a non-selected column

> *Determines the entire working set, that is data from all tables according to the JOINS ON clauses and also sub-queries*

Order 2 — WHERE clause

> *Filters the data according to the conditions*

Order 3 — GROUP BY clause

> *Aggregates the data according to one/more columns. Splitting data to different chunks or buckets, where each bucket has one key and a list of rows that match that key*

Order 4 — HAVING clause

> *Filters the aggregated data, that is it is used to filter out some buckets created by GROUP BY clause*

Order 5 — WINDOW functions

> *Just like grouping : performs calculation on a set of rows but each row will keep its own identity and won't be grouped into a bucket of other similar rows*

Order 6 — SELECT clause

> *Selects data after filtering & grouping. Use column names, aggregations and sub-queries inside it*

Order 7 — DISTINCT keyword

> *Discard rows with duplicate values from remaining rows left after the filtering and aggregations*

Order 8 — UNION keyword

> *Combines the result sets of two queries into one result set*




> *NOTE: using DISTINCT prevents sorting by a non-selected column*

Order 10 — LIMIT clause

> *Discard all rows but the first X rows of the query's result*

. . .

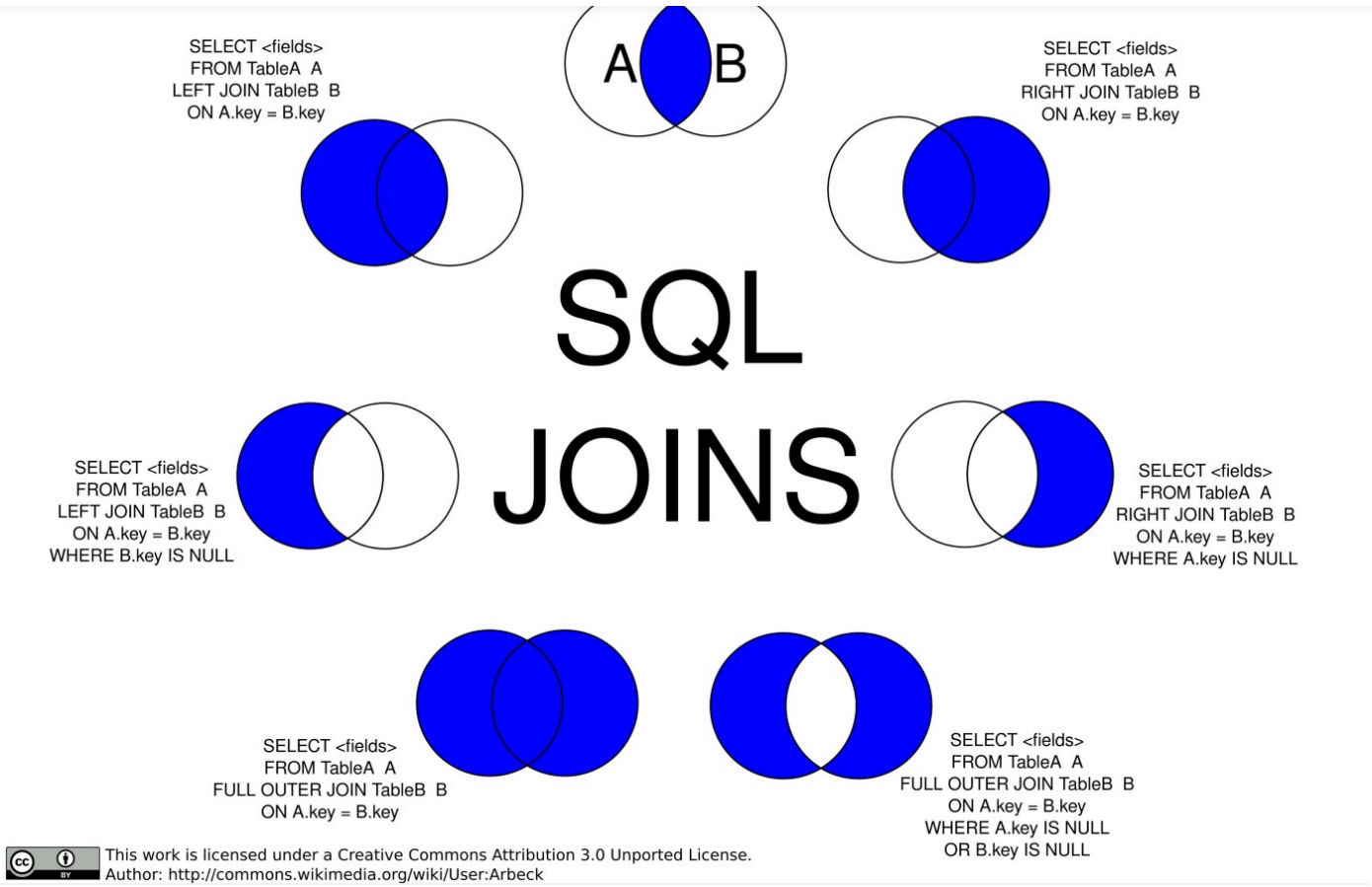
3. JOINs vs SETs — practical analogy to differentiate

Hoping you're familiar with varied JOIN statements, I want to set a very practical analogy on how to differentiate between JOINs and SETs.

JOINs — Visualize tables to be CIRCLEs

A circle has only 1 dimension — radius.
That is, while performing JOIN statement, we only need *a common field* (radius) to combine data or rows from *two or more tables* (circles).



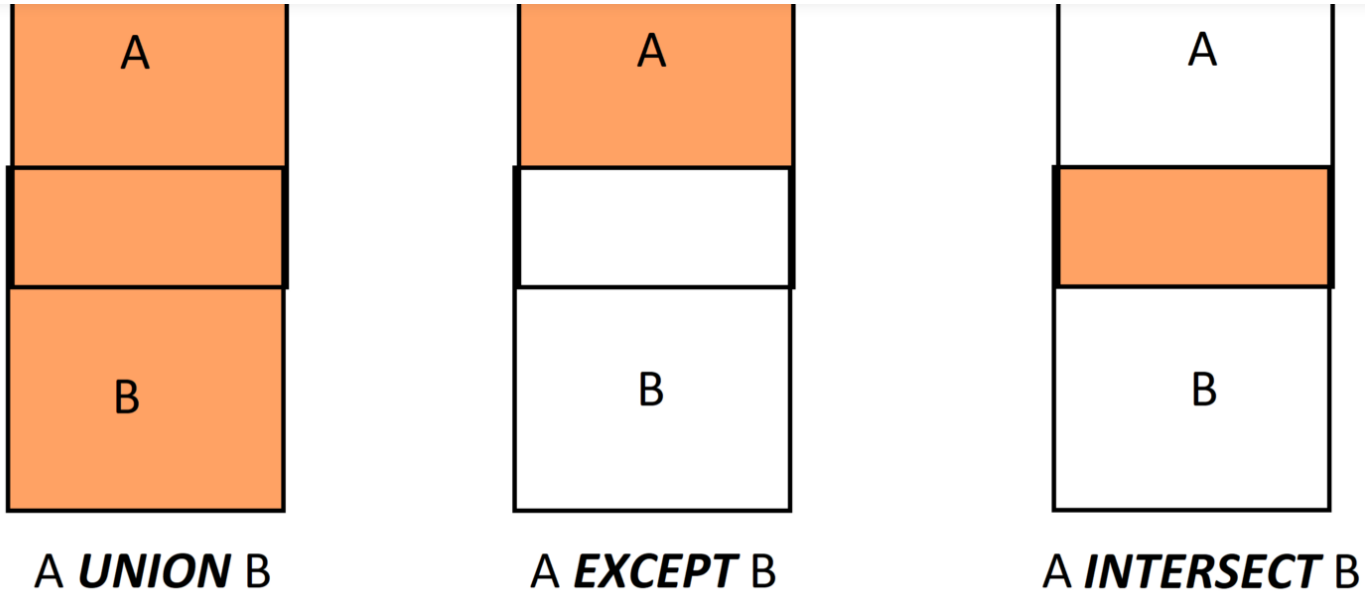
Arbeck / CC BY (https://creativecommons.org/licenses/by/3.0)

Note: Do Not Confuse SQL JOINs Visual Analogy to Venn Diagrams

SETs — Visualize tables (from select statements) to be RECTANGLEs

A rectangle has 2 dimensions — length and breadth.

That is, while performing SET statement, *fields* (length and breadth) used in the *select statements* (rectangles) must be in the same order, same number and same data type, to combine and produce the result set as UNION, UNION ALL, EXCEPT or INTERSECT of separate select statements.



- Note the pictorial ORIENTATION of the tables in above ENSURES *fields* are in the same order, same number and same data type.
- The UNION clause produces distinct values in the result set. To fetch the duplicate values use UNION ALL.

. . .

4. Sub-queries — A Query within Query

Scalar Sub-query — MUST return just one record

Sub-query in SELECT list:

```
SELECT
    Category.CategoryName,
    ( SELECT MAX(DVDPrice) FROM Films
            WHERE Films.CategoryID = Category.CategoryID ),
    Category.CategoryID
FROM Category
```

- And because of this **LINK** — aggregate function `MAX(DVDPrice)` — **returns only one value**, that is the max. price for each category in the `Category` table

Sub-query — returning more than 1 value

Sub-queries are very powerful when used in WHERE clause and/or coupled with IN, NOT IN, ANY / SOME, ALL, EXISTS, NOT EXISTS comparison conditions.

ANY or SOME

```
SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
```

- Return `TRUE` if the comparison of ***s1*** is `TRUE` for `ANY` of the values in the column or **list of values** that the sub-query returns
- ANY must follow **a comparison operator** (=, >, <, >=, <=, <>, !=), that is between the column name `s1` and comparison condition `ANY`

E.g.: Suppose that there is a row in `t1` containing `(10)` then the expression is:

- `TRUE` if `t2` contains `(21,14,7)`
- `FALSE` if `t2` contains `(20,10)` or if `t2` is empty
- *Unknown* (i.e. `NULL`) if `t2` contains `(NULL,NULL,NULL)`

ALL

```
SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);
```

- Return `TRUE` if the comparison ***s1*** is `TRUE` for `ALL` of the values in the column or **list of values** that the sub-query returns

E.g.: Suppose that there is a row in `t1` containing `(10)` then the expression is:

- `TRUE` if `t2` contains `(-5,0,5)` or if table `t2` is empty
- `FALSE` if `t2` contains `(12,6,NULL,-100)`, as 12 > 10
- *Unknown* (i.e. `NULL`) if `t2` contains `(0,NULL,1)`

IN and NOT IN

```
SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2);
```

- Returns `1` (true) if ***`s1`*** is **equal to any** of the values in the `IN()` **list** that the sub-query returns, else returns `0` (false). NOT IN reverses the logic
- Does the execution of `IN` comparison condition sounds a little familiar — Yes, the keyword `IN` is an alias for `= ANY` .
  But, `NOT IN` is not an alias for `<> ANY`, but for `<> ALL`
- **No comparison operators** (=, >, <, >=, <=, <>, !=) are allowed between column name `s1` and comparison condition `IN`

EXISTS and NOT EXISTS

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

- If a ***`subquery`*** (SELECT * FROM t2) returns any rows at all, then condition `EXISTS` ***`subquery`*** is `TRUE`, and `NOT EXISTS` ***`subquery`*** is `FALSE`
- That is, if `t2` contains any rows, even rows with nothing but `NULL` values, the `EXISTS` condition is `TRUE`




with correlation `cities_stores.store_type = stores.store_type`

```
SELECT DISTINCT store_type FROM stores
  WHERE EXISTS (SELECT * FROM cities_stores
    WHERE cities_stores.store_type = stores.store_type);
```

More Examples

. . .

References

- MySQL 8.0 Reference Manual
- A blog written by Tomer Shay @ EverSQL
- JOIN image Arbeck/CC BY https://creativecommons.org/licenses/by/3.0

. . .

Thank you for reading! Let me know in a comment if you felt like this did or didn't help. If this article was helpful, *share it*.

LinkedIn

Eklavya Saxena - Greater New York City Area | Professional Profile | LinkedIn

Proficient data analyst and aspiring data scientist with more than 2 years of industry experience in sales or customer...

www.linkedin.com/in/EklavyaSaxena