SQL Query – Cheat Sheet

After completing Master SQL for Data Science (https://www.udemy.com/master-sql-for-data-science/), from Imtiaz Ahmad on Udemy, which I **highly** recommend, I felt the need to create a cheat sheet for myself so I'll have the basics to hand when memory fails – which it inevitably does in the beginning… This document does not explain how everything works, it's literally just quick reminders so ***do the course first***, and then make use of this next!

A few general comments to begin:

'Single quotation marks' are required for most occasions!

Each statement is followed by a semi-colon; but if you only have one statement then this is irrelevant.

SQL is not a case-sensitive language but by convention commands, functions and keywords appear in upper-case and fields, tables, views, etc. will appear in lower-case.

Putting data in

SQL Script	Notes

`CREATE TABLE table_name (`
`  field1 varchar(100),`
`  field2 varchar(100),`
`  primary key (field1)`
`);`
	Create a new table called table_name with 2 fields: field1 and field2. The primary key will be the unique identifier that is always present. Examples of data types are variable characters, date, integer, etc.

`INSERT INTO table_name VALUES ('Cape Town','Western Cape');`
`INSERT INTO table_name VALUES ('Hermanus','Western Cape');`
`INSERT INTO table_name VALUES ('Queenstown','Eastern Cape' );`
	Add new rows to the table with values as specified

Basic data selection

SQL Script	Notes
`SELECT * FROM table_name;`	Select all fields from the specified table

`SELECT firstname, lastname, email`
`FROM table_name`
`WHERE region = "Western Cape";`
	Select only the fields specified from the table specified *where* specific criteria are met

`* - full wildcard`	

`% - partial wildcard, for example:`
`SELECT first_name, last_name, email`
`FROM table_name`
`WHERE first_name like '%ind%';`
	Select first names that contain 'ind' with any characters before or after, e.g. Cinderella, Linda, etc.

`SELECT *`
`FROM employees`
`WHERE 1=1;`
	Twisted logic – it will always bring you all results!

`SELECT *`
`FROM table_name`
`WHERE first_name = Linda`
`AND salary > 500000;`
	Selection with multiple conditions – AND
	Returns all the well-paid Lindas

`SELECT *`
`FROM table_name`
`WHERE first_name = Linda`
`OR salary > 500000;`
	Selection with multiple conditions – OR
	Returns people named Linda or people who earn well

SQL Script	Notes
SELECT * FROM table_name WHERE salary < 500000 AND (first_name = 'Linda' OR first_name = 'Carmen') ;	Combining AND's and OR's requires brackets
SELECT * FROM table_name WHERE first_name IN ('Linda', 'Carmen', 'Julia') ;	With multiple OR's it is more practical to use the IN keyword
SELECT * FROM table_name WHERE salary BETWEEN 400000 AND 500000;	With multiple numeric values we can use ranges rather
SELECT first_name, birth_date FROM table_name WHERE birth_date BETWEEN '1970-01-01' AND '1990-01-01';	Notice that with multiple date values single quotation marks are required
SELECT * FROM table_name WHERE NOT first_name = 'Linda; *-- or otherwise like this* SELECT * FROM table_name WHERE first_name != 'Linda;	Various ways of saying NOT Note that comments that are not executable can be preceded with --, a double-dash
SELECT * FROM table_name WHERE first_name IS NULL; SELECT * FROM table_name WHERE NOT first_name IS NULL;	Special select of null values from a field Or selection of non-null values from a field
-- Sort ascending SELECT * FROM table_name WHERE NOT first_name IS NULL ORDER BY region; -- Sort descending SELECT * FROM table_name WHERE NOT first_name IS NULL ORDER BY region DESC;	Sort fields once selected, in the specified order – ascending by default (or with asc), otherwise descending (with desc)
SELECT DISTINCT first_name FROM table_name ORDER BY first_name;	Get a list of unique first names from your data, sorted alphabetically
SELECT * FROM table_name LIMIT 10;	Just get the first 10 records – a quick way to visualize the data
-- Alternate field names with no spaces SELECT DISTINCT first_name as unique_first_name FROM table_name ORDER BY unique_first_name -- Alternate field names with spaces SELECT DISTINCT first_name as "First Name" FROM table_name ORDER BY "First Name";	Columns are given default names at run-time but we can re-name columns as required using the AS keyword. Note the use of double quotation marks in this case!

Formatting and manipulating the data

SQL Script	Notes
SELECT UPPER(first_name) FROM table_name;	Function to display all data in upper case
SELECT LOWER(first_name) FROM table_name;	Function to display all data in lower case

SQL Script	Notes
-- Rounding to a set number of decimal places SELECT ROUND(price * 1.15, 2) as price_incl_vat FROM table_name; -- Rounding to the default number of decimal places (0) SELECT ROUND(price * 1.15) as price_incl_vat FROM table_name;	Function to round the resulting numbers, either to 0 decimal places by default or to the set number desired
SELECT LENGTH(first_name) FROM table_name ORDER BY length DESC;	Function to return the *length* of each field entry, e.g. this could find us the longest number of letters used for the first_name field
SELECT POSITION('@' IN email) FROM table_name;	Function to return which position the @ sign occurs in
SELECT TRIM(first_name) FROM table_name;	Function to get rid of that pesky white space people sometimes leave at the beginning and/or end of their data
SELECT LENGTH(TRIM(first_name)) FROM table_name ORDER BY length DESC;	Functions can live inside other functions – for example this statement returns the length of first name *after* trimming
SELECT first_name || ' ' || last_name as "Full Name" FROM table_name;	Concatenation is achieved with || - add extra between values in single quotation marks
-- Displaying the first 5 chars SELECT SUBSTRING(first_name FROM 1 FOR 5) FROM table_name; -- Displaying character 5 onwards SELECT SUBSTRING(first_name FROM 5) FROM table_name;	Function to display only n characters of the string
SELECT SUBSTRING(email FROM POSITION('@' IN email) + 1) FROM table_name;	So with a function inside a function, we can split our data to get “only domain names from email addresses”
SELECT SUBSTRING(email FROM 1 FOR POSITION('@' IN email) - 1) FROM table_name;	Or we can split our data to get “only usernames from email addresses”
SELECT first_name, last_name, (‘Western Cape’ IN (region)) as region_wc FROM table_name;	Adds a *third column* which gives true if the region is Western Cape, otherwise false
SELECT first_name, REPLACE(first_name, 'Linda', 'Linda Lye') FROM table_name;	Adds a *second column* which gives the intended replacement text for each first_name column
SELECT COALESCE(first_name, 'NONE') FROM employees;	Adds a *second column* which gives all the first names that were there + NONE where the first_name column contained a NULL value

Aggregate functions

SQL Script	Notes
SELECT MAX(salary) FROM table_name;	Maximum
SELECT MIN(salary) FROM table_name;	Minimum
SELECT AVG(salary) FROM table_name;	Average
-- Select count of a single field SELECT COUNT(salary) FROM table_name; -- Select count of records SELECT COUNT(*) FROM table_name;	Count non-null records
SELECT SUM(salary) FROM table_name;	Sum

Group By

Where aggregate functions are used in conjunction with *other* fields, GROUP BY is used to group the data so that the aggregate still makes sense. Obviously, the fields to group by themselves should make sense e.g. to show the sum of salaries by first name would be nonsensical, but to show the sum of salaries by region would make sense!

SQL Script	Notes

`SELECT region, AVG(salary)`
`FROM table_name`
`GROUP BY region;`
	Returns average salary by region. The number of records will be equal to the number of regions

`SELECT region, COUNT(*) total_employees, ROUND(AVG(salary)) avg_salary, MIN(salary) min_salary, MAX(salary) max_salary`
`FROM table_name`
`WHERE salary > 500000`
`GROUP BY region`
`ORDER BY avg_salary desc;`
	We can build in additional functions, so the query alongside re-names all the aggregate fields and then sorts the results in descending order of average salary

`SELECT municipality, district, region, MAX(salary)`
`FROM table_name`
`GROUP BY municipality, district, region;`
	In this example we have 3 non-aggregate fields, and therefore the same 3 fields in GROUP BY

`SELECT region, count(*)`
`FROM table_name`
`GROUP BY region`
`HAVING count(*) > 100`
`ORDER BY region;`
	Filtering on aggregated data is not done with WHERE but rather with HAVING. In this example, after obtaining the count of records per region we then only display those where the count was more than 100, smaller regions will be omitted.

Sub-queries and views as data sources

Queries can exist within queries. Always read from the inside out to work out what is going on. In general, it looks like it would be easier to create views for complicated subqueries as these can then easily be referenced by name and you don't have to have all those pesky wheels within wheels...

SQL Script	Notes

`SELECT data.first_name`
`FROM (SELECT first_name, last_name, region`
`FROM table_name) data;`
	Here we have a typical SELECT statement within another SELECT statement. The inner select statement has been given the alias data and becomes the data source of the outer select statement

`SELECT first_name, salary` `FROM (SELECT * FROM table_name WHERE salary > 500000) as set;`	Sub-queries can be used in the FROM clause. In this case an alias must always be assigned.

`SELECT DISTINCT(region)`
`FROM table_name`
`WHERE region NOT IN (SELECT region FROM table_regions);`
	Sub-queries can also be used in the WHERE clause. In this example we find all regions in table table_name that do not exist in table_regions, a useful data integrity check

`SELECT first_name, region_id, salary - (SELECT MAX(salary)`
`FROM employees) shortfall`
`FROM employees;`
	Sub-queries can also be used in the SELECT clause. In this case only one row of data should be returned by the sub-query e.g. MAX(salary) alongside.

`SELECT known_as, curr_salary` `FROM (SELECT first_name known_as, salary curr_salary FROM table_name WHERE salary > 500000) as set;`	If columns are re-named in the sub-query then that is how they must be referenced in the outer query

`CREATE VIEW v_people_info as`
`SELECT first_name, region_id, salary - (SELECT MAX(salary)`
`FROM employees) shortfall`
`FROM employees;`
	Create a view for the entire query. This view can then be referenced just like a table by name. The v_* naming convention is traditional!

`-- Look for people whose salary is greater than`
`-- the regional average`
`SELECT first_name, salary`
`FROM table_name t1`
`WHERE salary > (SELECT ROUND(AVG(salary))`
`FROM table_name t2`
`WHERE t1.region = t2.region);`
	Correlated sub-queries which make use of aliases for differentiation, are also possible, but very expensive because for each record of the outer query, the inner query has to run which means with 1000 records in the outer query the inner query will have to run 1000 times in order to return a result!

Boolean logic

SQL Script	Notes
`= != <> > <`	Equals, Not equal, Not equal, Greater than, Less than

`-- In a table with 30 records, the following returns the 1st 9`
`SELECT running_number`
`FROM table_name`
`WHERE running_number < ALL (`
`  SELECT running_number`
`  FROM table_name`
`  WHERE running_number > 9`
`);`
`-- In a table with 30 records, the following returns none`
`SELECT running_number`
`FROM table_name`
`WHERE running_number > ALL (`
`  SELECT running_number`
`  FROM table_name`
`  WHERE running_number > 9`
`);`
	< ALL or > ALL can be used as a Boolean operator against a set returned by another SELECT statement.

`-- In a table with 30 records, the following returns all`
`SELECT running_number`
`FROM table_name`
`WHERE running_number < ANY (`
`  SELECT running_number`
`  FROM table_name`
`  WHERE running_number > 9`
`);`
`-- In a table with 30 records, the following returns 11 onwards`
`SELECT running_number`
`FROM table_name`
`WHERE running_number > ANY (`
`  SELECT running_number`
`  FROM table_name`
`  WHERE running_number > 9`
`);`
	< ANY or > ANY can be used as a Boolean operator against a set returned by another SELECT statement.

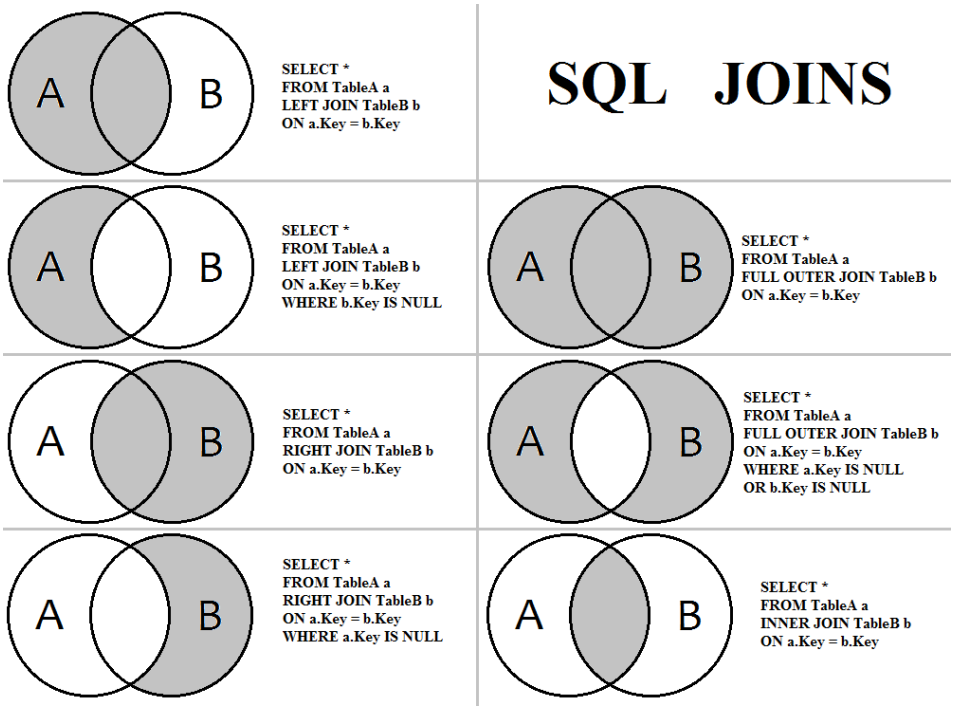
Conditional expressions

SQL Script	Notes

`SELECT first_name, salary,`
`CASE`
`  WHEN salary < 200000 THEN 'Under-paid'`
`  WHEN salary >= 200000 THEN 'Over-paid'`
`  ELSE 'Not determined'`
`END payment_status`
`FROM table_name;`
	The CASE clause let's you create a new column and fill its contents conditional on various conditions being met (WHEN / THEN) with a catch-all (ELSE) as optional but good practice.

Joins

The diagram…



SQL Script	Notes

`SELECT first_name, region`
`FROM table_name, table_regions`
`WHERE table_name.region_id = table_regions.region_id;`
	A simple join between 2 tables

`SELECT first_name, country, category`
`FROM table_name t INNER JOIN table_regions r`
`ON t.region_id = r.region_id`
`INNER JOIN table_categories c`
`ON t.category_id = c.category_id;`
	A more typical syntax – INNER join specified, you can daisy chain these up

`SELECT DISTINCT table_names.region, table_regions.region`
`FROM table_names LEFT OUTER JOIN table_regions`
`ON table_names.region_id = table_regions.region_id;`
	An example of LEFT OUTER JOIN, all single values from table_names and matching values from table_regions

`SELECT DISTINCT table_names.region, table_regions.region`
`FROM table_names RIGHT OUTER JOIN table_regions`
`ON table_names.region_id = table_regions.region_id;`
	An example of RIGHT OUTER JOIN, all single values from table_regions and matching values from table_names

`SELECT DISTINCT table_names.region, table_regions.region`
`FROM table_names FULL OUTER JOIN table_regions`
`ON table_names.region_id = table_regions.region_id`
`WHERE table_names.region IS NULL`
`OR table_regions.region IS NULL;`
	An example of FULL OUTER JOIN, all single values from table_names and all single values from table_regions – but then by specifying we only want null values we get the small subset of values where there are null values in either set.

Some set type clauses

SQL Script	Notes

`SELECT region`
`FROM table_names`
`UNION`
`SELECT region`
`FROM table_regions;`
	This query PLUS that query, stacked up on top of one another with unique values (duplicates removed)

SQL Script	Notes
SELECT region FROM table_names UNION ALL SELECT region FROM table_regions;	This query PLUS that query, stacked up on top of another with all values (no duplicates removed)
SELECT region FROM table_names EXCEPT SELECT region FROM table_regions;	This query MINUS that query – all the results from the first query except for those that occurred in the second query. In fact in the Oracle environment the MINUS keyword is used in lieu of EXCEPT!

Window functions using OVER()

SQL Script	Notes
select first_name, region, COUNT(*) OVER(PARTITION BY region) FROM table_name;	In this scenario you get *all* records from table_name and a 3rd column which gives you the count of records by region
select first_name, region, AVG(salary) OVER(PARTITION BY region) FROM table_name	You can use different aggregate functions, as we have seen so far. In this scenario you get *all* records from table_name and a 3rd column which gives you the sum of salaries in each region
select first_name, region, COUNT(*) OVER(PARTITION BY region) ppl_region, salary, AVG(salary) OVER(PARTITION BY region) avg_salary FROM employees ORDER BY region	And because this is essentially adding a field each time, you can add other fields at will
SELECT first_name, hire_date, salary, SUM(salary) OVER(ORDER BY hire_date) running_total FROM table_name -- In longhand, the above actually means SELECT first_name, hire_date, salary, SUM(salary) OVER(ORDER BY hire_date RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) running_total FROM table_name	Using ORDER BY within OVER you can get a new column with a running total. In this scenario you get *all* records from table_name and a 3rd column which gives you the sum of salary in this row + the preceding row
SELECT first_name, hire_date, salary, SUM(salary) OVER(ORDER BY hire_date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) running_total FROM table_name	This means you can specify various values for the number of preceding rows to include in your operation, e.g. in this example, for each row the value in the preceding 1 rows is added. You could vary how many “PRECEDING” you include, e.g. 3 PRECEDING, etc.
SELECT first_name, hire_date, region, salary, SUM(salary) OVER(PARTITION BY region ORDER BY hire_date) running total FROM table_name	And using a combination or PARTITION BY and ORDER BY within OVER you can get a new column with a running total that resets each time your PARTITION BY value changes
SELECT first_name, region, salary, RANK() OVER(PARTITION BY region ORDER BY salary DESC) FROM table_name	Here we are ranking by salary in descending order, with a reset each time region changes
SELECT * FROM (SELECT first_name, region, salary, RANK() OVER(PARTITION BY region ORDER BY salary DESC) FROM table_name) data_source WHERE rank = 1	These types of queries can also be used as inline queries, so in the example alongside we only wanted to see the top-ranking earners in each region
SELECT * FROM (SELECT first_name, region, salary, NTILE(5) OVER(PARTITION BY region ORDER BY salary DESC) salary_bracket FROM table_name) data_source	Here we are splitting our data into n groups, so 5 buckets or 10 buckets

SQL Script	Notes
SELECT first_name, region, salary, FIRST_VALUE(salary) OVER(PARTITION BY region ORDER BY salary DESC) FROM table_name -- Another way to do the same thing is: SELECT first_name, region, salary, MAX(salary) OVER(PARTITION BY region ORDER BY salary DESC) salary_bracket FROM employees	Here we are making a 4th column which will contain the first value in salary (which has been sorted descending and thus will provide the top salary) and will reset each time region changes. It seems like you have to be very sure that FIRST_VALUE will always correctly meet your criteria, in this case I think MAX would just be safer?!
SELECT first_name, region, salary, NTH_VALUE(salary, 5) OVER(PARTITION BY department ORDER BY salary ASC) salary_bracket FROM table_name	Along with FIRST_VALUE we have the companion NTH_VALUE where you can specify which number value you want to pull out
SELECT first_name, salary, LEAD(salary) OVER() next_row_salary FROM table_name	In this scenario you get *all* records from table_name and a 3rd column which gives you the salary in the *next row* so think of LEAD giving you a headstart
SELECT first_name, salary, LAG(salary) OVER() next_row_salary FROM table_name	In this scenario you get *all* records from table_name and a 3rd column which gives you the salary in the *previous row*
SELECT first_name, region, salary, LEAD(salary) OVER(PARTITION BY region ORDER BY salary asc) next_higher_salary FROM table_name	In this scenario we'd use LEAD to give us a 4th column with the *next highest* salary in the region compared to the current record

Thanks for reading ☺