**SQL BASICS**

SQL stands for “**Structured Query Language”**.

It is the language used to communicated with databases, this includes *retrieving data* (known as querying) but also inserting, updating, or deleting data as well as creating or changing the database themselves.

SQL is the standard language for **relational database management systems (RDBMS)**

When coding in SQL, various clauses must be written in a *specific order* 🡪 **Logical Sequence** or **Syntax Sequence**

The logical sequence for the SELECT function is:

1. SELECT
2. DISTINCT
3. FROM
4. WHERE
5. GROUP BY
6. HAVING
7. ORDER BY

If you do not follow this order, then the query will fail

The logical sequence is not the order in which the clauses is *processed*, that is known as the **Processing Order** which looks like this:

* FROM
* WHERE
* GROUP BY
* HAVING
* SELECT
* DISTINCT
* ORDER BY

This affects how we write queries

We will now go over the basic clauses and keywords present in an SQL query

CREATING A DATABASE

To create a database, simply query CREATE DATABASE name, make sure you do this in the master database

This will allow you select the database from the select database tab

To create a table:

* When experimenting you can put DROP TABLE IF EXISTS to ensure there are no errors when editing the table
* Create a table using the CREATE TABLE clause followed by the name of the table, and then the name of the data in the table including its data type

A close-up of a white background

Description automatically generatedTo create a PRIMARY KEY

* At the end of the column that you want to be the PK, write INT PRIMARY KEY IDENTITY (1,1)

The IDENTITY (1,1) clause will create a unique value that starts at 1 and increments by 1 each time, this means we do not have to input the PK every time we add data

To create a FOREIGN KEY

* Select the column from the FACT TABLE that is referenced in another DIMENSION TABLE
* Next to its data type write FOREIGN KEY REFERENCES table\_name

To input data into a table:

* Start with INSERT INTO table\_name and then identify what columns you will be adding data into
* Then use the VALUES clause to input the data for each column you just identified, this should match the data type specified in the CREATE TABLE clause

To alter the content in a table (without dropping the table and remaking it)

* Use the clause ALTER TABLE table\_name
* Follow this by another clause of your choice:
  + ADD column\_name datatype
  + DROP COLUMN column\_name
  + RENAME COLUMN old\_name TO new\_name
  + ALTER COLUMN column\_name new\_data\_type
  + ADD CONSTRAINT new\_column\_name FOREIGN KEY (chosen\_column)

SELECT

The SELECT clause specifies *which* columns we want to run a query on – simply provide a *comma separated list* of column names

You can additionally change the way that the column headings are presented in the *output* – this is to improve readability and presentation

* Good practice to use the AS keyword for column aliases to help separate it from the column definition
* Make sure to use “**double quotes”** instead of single quotes



If you want to return *all* columns in a table, then you can use an *asterisk*

WHERE

The WHERE clause is used to only return *specific* ***rows*** from our tables, it will only return data with the conditional statement

* When your specification is a *string,* you should use **‘single quotes’**

A screen shot of a computer

Description automatically generated

There are several **comparison operators** that you can use when creating your conditional statement:

* < less than
* > more than
* <= less than or equal to
* >= more than or equal to
* != not equal to

The WHERE clause is not just limited to a *single comparison*, you can use AND to return only the rows for which *multiple* statements are true

You can also use OR to return rows that match any or all the statements

* Important to note that you have to specific the row in each AND/OR statement regardless of if it is from the same row

A black background with white text

Description automatically generatedSometimes you may not want to return *exact matches*, this is when **wildcards** come in use as they can be act as *substitutes* for other characters

* \_ **underscore** can be used to substitute a *single character*, e.g., ‘Li\_a’ the underscore can be any letter
* % **percentage** can substitute *zero or more characters* e.g., ‘J%’ will find any string beginning with J but any characters after
* []**square brackets** are used to specify multiple possible character to match e.g., ‘Kindle Fire [567]’ would return results for those with the numbers specified
* [^ ]**square brackets and up arrow** is used negate characters in the square brackets e.g., ‘Kindle Fire [^1234]’ would not return any of the specified characters

These can be combined within a single string e.g.,

A black rectangular object with white text

Description automatically generatedThis returns anything beginning with A, B, or C

A black background with white text

Description automatically generatedThis returns anything that has ‘HD’ in the text regardless of what’s before or after it

BETWEEN

BETWEEN can be used when specifying a *range* of numbers

A black screen with numbers

Description automatically generatedIt may also be used alongside other conditional statements

A black background with white text

Description automatically generated

IN

In some cases, you will want to exactly match the number of possibilities, when there are multiple you can use the OR clause

However, this may not be suitable when there are many possibilities, the IN clause is more suitable here

The IN keyword returns rows with an entry that matches any of the things were specify within a list which is **(enclosed in round brackets)**

* Each item in the list is surrounded by its own **single quotations**

A black background with white text

Description automatically generated

NULL

NULL refers to when some of the data in the database is missing, does not exist, or N/A

It behaves in an unfamiliar way as it is a **non-value**:

* NULL is not == 0
* NULL is not == ‘ ‘
* There is *nothing* that is equal to NULL
* Even NULL is not equal to NULL

To find missing values, you cannot say that the specific value is *equal to (=)* NULL due to the above characteristics

Instead, you must look for entries where the information IS NULL

To find data that *does not* contain NULL entries, you use IS NOT NULL

A black rectangular object with a white border

Description automatically generated

TOP & ORDER BY

TOP keyword is used when we want to determine what the contents of a table *looks like* as we can display the first few rows instead of the entire table



ORDER BY is the last clause processed by SQL, it sorts the results based on the calculations that we specify – you can specify ascending (ASC) or descending (DESC) order

* By default, it will return in *ascending order*

A screen shot of a computer

Description automatically generatedYou can use ORDER BY and TOP together to grab the first few rows in ASC or DESC order rather than arbitrarily

DISTINCT

The DISTINCT keyword allows us to remove *duplicate rows* from queries

A black background with white text

Description automatically generated

CONCATENATION

This is not a clause, but rather a means to *connecting things together* – lets us combine text from multiple columns

The way that you do this depends on the SQL version being used:

* SQL server 🡪 put a + symbol *between* the columns or text you want to combine
* ANSI SQL 🡪 put two || symbols *between* the columns or text you want to combine

You must use *aliases* when conducting concatenation otherwise the column header will be ‘No column name’

A black screen with white text

Description automatically generated

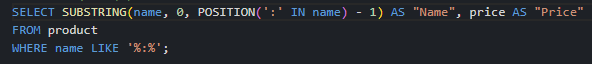
STRING FUNCTIONS

SQL has built-in functions for use in further manipulation of text in queries

To use *functions* in SQL, you must specify the function name followed by one or more parameters (in brackets)

* UPPER() returns the contents of that text in *uppercase*
* LOWER() returns the contents of that text in *lowercase*
* TRIM() removes *whitespace characters* at the start or end of a string
* LTRIM() removes *whitespace* at the start (left side) of a string
* RTRIM() removes *whitespace* at the end (right side) of a string
* LENGTH() returns the length of a string – in SQL server this function is LEN()
* LEFT(text, N) returns the leftmost N characters of a string
* RIGHT(text, N) returns the rightmost N characters of a string
* SUBSTRING(text, start index, length) returns several characters equal to *length*, starting from the *start index* from a string
* POSITION(substring IN text) returns the index (position) of the first instance of the character within the text, or a 0 if that character is not within the text
* CHARINDEX(character to look for, N, start index \*optional) finds the position of a substring or expression in each string, it returns the first position and ignores the rest

These functions can be combined in all sorts of ways, functions within functions are known as **Nested Functions**

A screenshot of a computer screen

Description automatically generatedThe above code returns the first letter of their first name and then capitalises the first letter of the surname followed by lowercase

The above code involves several clauses and involves some nested functions

* In this instance, the substring starts at index 0 and continues until it reaches the position of : - 1 i.e., the character *before* the :
* WHERE name is used to remove all rows that do not contain :

A screen shot of a computer

Description automatically generated

ARITHMETIC

We can manipulate numerical data using arithmetic operators:

* Addition: +
* Subtraction: -
* Multiplication: \*
* Division: /
* Modulo: % this returns the *remainder* after *division* 
  + E.g., 17 % 5 =2, 5 goes into 17 three times to produce 15, the difference between 17 and 15 is 2 (the remainder)

A screenshot of a computer

Description automatically generatedAn example or using modulo is:

DATE FUNCTIONS

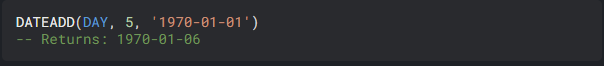
These are functions for when working with dates

SQL handles dates according to international standard formats: YYYY-MM-DD, when time is included, this format is: YYYY-MM-DD hh:mm:ss.s

* GETDATE() returns the current date
* DATEADD(unit, N, date) adds on to the date provided a number (N) of the unit specified
* DATEDIFF(unit, date1,date2) returns the difference between two dates
* YEAR(date) extract the year as an integer from the day
* MONTH(date) extracts the month as an integer from the data
* A black background with numbers and letters

  Description automatically generatedDAY(date) extracts the day as an integer from the data

A black background with numbers and letters

Description automatically generatedBelow code demonstrates the use of date functions by showing the year in which individuals were 13 years and 9 months old

A screenshot of a computer

Description automatically generated

CASE

CASE is used when we want a different calculation or output depending upon the value for each row

CASE begins a *case statement*, each condition is specified using WHEN, with the outcome specified using THEN

* These conditions are processed in order

Any conditions that are not specified will fall in the ELSE condition

Finally, must use END to close the case statement, this is also where we define the alias

A screen shot of a computer program

Description automatically generatedAbove creates a new column of STOCK LEVEL as completely different conditions based on how much stock is left

AGGREGATION

Aggregation is the act of ***bringing things together into groups***

We can aggregate an entire table into a single number

A black screen with orange text

Description automatically generatedUnlike other functions, this brings back only a singular row

We can use multiple aggregate functions in a single query, each will produce its *own column*

Aggregation itself is not a function, but refers to any function that does bring things together into groups e.g., SUM

The above code takes the sum of price multiplied by stock to give the total amount of value from all available stock

* Important to do the price \* available stock within the same brackets, rather than separate ones as this will not yield the correct result

GROUP BY

GROUP BY is a clause that groups information together and is almost always used in conjunction with aggregate functions

It allows us to aggregate a specified column with which to group our table

A screenshot of a computer

Description automatically generatedThis calculates the average of the price column for each category for each category ID, grouping products in the same category *together*

Can also group by multiple columns, just must list them separated by multiple columns

An important rule:

* Everything in the SELECT clause must be an *aggregate* or appear in the GROUP BY clause
* I.e., everything in the SELECT clause must be aggregated or grouped

A screenshot of a computer program

Description automatically generatedLet’s break this down:

* SELECT city AS “City” means that the city names will be included in the final output
* COUNT(city) counts the amount of people in the cities and produces its own row for it
* GROUP BY city means the data will then be presented as grouped by the city name
* ORDER BY means the data will be ordered the amount of people in each city starting with the largest count (DESC)

HAVING

A screenshot of a computer

Description automatically generatedThe HAVING clause is used when we want to filter based on the result of aggregation

This returns the category IDs and their average prices when the average price is less than 200

* The aggregate must be *recalculated* cannot simply refer to it as its alias

HAVING and WHERE are different!

* WHERE 🡪 filters rows in the *original table*
* HAVING 🡪 used to filter *based on aggregation*

But it is possible to use both together

A screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedThe above code returns a list of product category IDs and the highest product price for each category

* SELECT is selecting the category id as well as the maximum price
* WHERE will only produce results that are NOT NULL
* GROUP BY is grouping the output by product\_category\_id
* HAVING is filtering data based on where the minimum available stock is below 100
* ORDER BY is indicating that the output will be ordered by category id in ascending order, A-Z

JOIN

The JOIN keyword combines tables *together* by matching on *rows*

Two tables are specified and a *rule* for joining them together:

* A diagram of a diagram

  Description automatically generatedThis will usually be that values in a column in Table 1 should be the same as values in a column in Table
* Whenever there’s a match in these columns, the rows are combined

JOIN statements are a part of the FROM clause

It is important to know which is the ‘Left Table’ and which is the ‘Right Table’

* The **‘Left Table’** is the *first table specified* and we join the Right TO the Left

When we join tables, we might end up with multiple columns with the *same name*

* To resolve this, we use a full stop: **table.column** where **table** is the *name of the table* the *column* belongs to, and **column** is the name of the column itself
* It is not strictly necessary when dealing with columns with *unique names*, but it is good practice

**Types of JOINS**

**LEFT JOIN (or LEFT OUTER JOIN)** 🡪 returns ALL rows in the LEFT table, and only returns rows from the right table that *match* with rows in the left table

A black screen with white text

Description automatically generated**RIGHT JOIN (or RIGHT OUTER JOIN)** 🡪 will return ALL rows in the RIGHT table, and only returns rows from the left table that *match* with rows in the right table (NOTE. Right joins are rare, it is better to avoid using them unless there is a good reason for doing so)

A black screen with white text

Description automatically generated**INNER JOIN (or simply JOIN)** 🡪 will return rows from each table ONLY if there is a match in the other table

A black screen with white text

Description automatically generated**OUTER JOIN (or FULL OUTER JOIN)** 🡪 will return ALL rows from EACH table, *regardless* of whether there is a match with the other table

A black screen with white text

Description automatically generatedFor LEFT, RIGHT, and OUTER JOINS, whenever there is *not a matching row* in one of the tables, the missing values will be represented by NULL

TABLE ALIASES

When completing JOIN, typing out full names can be cumbersome, we can give our tables aliases to streamline the process

* Standard practice is to use the **initials** of the table name, adding letters in situations where *two* tables share the *same initials*

A black screen with white text

Description automatically generatedTable aliases are completed differently to row aliases, simply put the alias name *after* where you first refer to it

Then, when you later need to refer to that table you can just use the simplified alias

An example of this utilised in a LEFT JOIN is below:

A screen shot of a computer

Description automatically generatedThis means that *all* data on the left is processed, whilst on the right only rows that match with the left are processed

A screenshot of a computer screen

Description automatically generatedThis gives this output, rather than the last line being cut off because of the NULL value, the left join means that Nicole is still processed

A screenshot of a computer screen

Description automatically generatedWhen this is changed to an INNER JOIN, Nicole is cut off the output as that type of join will ONLY return rows that match BOTH tables

MULTIPLE JOINS

It is possible to perform multiple joins within a single query, simply add the next join immediately afterwards

* Each new join can join on the original table specified at the start of the FROM clause or any table that has already been joined on this table

A black rectangular object with white text

Description automatically generatedIn the above example, table 3 and table 2 share a column in common and so can be joined

A screen shot of a computer

Description automatically generatedThe above example shows many joins, all connecting to each other through a common PK/FK

A screen shot of a computer

Description automatically generatedThis query produces this output, all data presented come from different tables connected to each other through several joins

It is important to have a good understanding of the database you are querying when performing multiple joins, having an ERD makes this easier to comprehend

UNION JOIN

JOIN will join rows from different tables together *horizontally*, UNION can join columns together *vertically*

These columns will often have the same names, but they do not have to

A screenshot of a computer

Description automatically generatedThe above example has joined the first name and last name of both customers and employees

* UNION will not duplicate rows that appear in both tables
* To return a query that does not remove these duplicates use UNION ALL

Column aliases only need to be specified in the first table

A SELECT clause comes after the UNION clause as you may only want to union two out of several tables in your first SELECT

SUBQUERIES

**SUBQUERY** 🡪 a query within a query, used to provide data to the *outside* query

* They can be used in SELECT, FROM and WHERE clauses

**SELECT SUBQUERIES**

Subqueries within the SELECT clause return a single value, which is almost always an aggregation of some kind, this single value will be the same for every row

A screenshot of a computer

Description automatically generatedBecause it will always return the same value, SELECT subqueries are often used to return a reference value for *further calculation*

A screen shot of a computer

Description automatically generatedThe above examples shows how much more expensive each item is compared to the cheapest item

A black screen with white text

Description automatically generatedThe below example creates a new column that displays how much each item differs from the average price

**FROM SUBQUERY**

We can use subqueries in the FROM clause, this is useful if we want to perform further calculations on top of previous aggregations or calculations

A screen shot of a computer

Description automatically generatedThis example calculates the total price per category, we then find the average of these totals

* The subquery is processed before the main query, so the alias “total price” can be referenced in the first SELECT clause

You can also JOIN a subquery, instead of a table name, define the subquery. It is **mandatory** to use an alias for the subquery

A screen shot of a computer program

Description automatically generatedSubquery aliases are defined differently, you define it after you close the bracket, right before the ON keyword

A screen shot of a computer program

Description automatically generatedThe above example calculates the minimum average of available stock for categories

* The average query is created as a subquery so that it can then be referenced to find the minimum value of

**WHERE SUBQUERY**

**A black screen with white text

Description automatically generated**The WHERE subquery should return a single column, which can be used as a list of values, we can then filter our list based on this list, most commonly using IN or NOT IN

This subquery returns a list of product IDs for products that have been ordered.

By filtering the results to only include products with an ID in that list, we can see the product name for ordered products without *joining tables*

*A screen shot of a computer

Description automatically generated*The example above shows how we do not need to join tables when using a subquery, this query returns customers name and phone number if they have *never purchased something before*

Another way to do this is to use NOT EXISTS, SELECT 1 simply selects the first row, this means we must input another where clause

SELECT DISTINCT firstname AS "First Name", phone\_number AS "Phone Number"

FROM customer c

WHERE NOT EXISTS (

    SELECT 1

    FROM purchase\_order po

    WHERE po.customer\_id = c.customer\_id) AND

    phone\_number IS NOT NULL;

STORED PROCEDURES

You may find yourself writing the same query but with different parameters, this can be very repetitive

* We can define a *procedure* with variables = **STORED PROCEDURES**

A stored procedure is a reusable SQL query which is saved within the database server

* They have the capability to *accept parameters* thus allowing us to *modify* the behaviour of the query

The benefits of using stored procedures include:

* **REUSABILITY** 🡪 we do not need to keep writing the same SQL query logic repeatedly
* **PERFORMANCE** 🡪 when a stored procedure is created, the database compiles and optimises the way it processes stored procedure queries
* **SECURITY** 🡪 stored procedures are usually parametrised which is an extra level of protection against potential SQL injection attacks. Moreover, we can grant permissions to those who are able to access stored procedures which adds an extra layer of security

The below syntax is T-SQL, other RDBMS may use a different syntax called SQL/PSDM (Persistent Stored Module)

**CREATING A STORED PROCEDURE**

If we wanted to create a SP to update the Address for Customers based on their ID, then this could be written as:

A screenshot of a computer program

Description automatically generatedA stored procedure is first creating using the CREATE PROCEDURE command, it is then:

* Defining two *parameters* called @ID and @NewNumber
* Formulating the query logic to:
  + Locate the customer based on their Customer\_ID specified by the @ID parameter
  + Update this customer’s Phone\_Number using the input parameter @NewNumber
* Return 0 indicates successful execution. The return value can be captured by a calling program.

A screenshot of a computer code

Description automatically generatedA breakdown of this process can be seen here:

In a SQL server environment, to access stored procedures, navigate through our RDBMS interface by following this path Programmability > Stored Procedures

**EXECUTING A STORED PROCEDURED**

Now whenever we want to update a customer’s number, we can just execute the stored procedure, and provide the appropriate parameters

VIEWS

Some queries can be quite complex, this means it can be become tedious to re-write the same SQL statement every time you require it

We can save our query to something called a VIEW which provides us with a more efficient and convenient way to manage complex querying scenarios.

A view is a virtual table – the *instructions* to create the view table are stored in the database, rather than an actual table

We can create a view using the CREATE VIEW statement

A screen shot of a computer

Description automatically generatedThis query counts the number of products in each category the query has been saved using the CREATE VIEW statement which will be saved within the RDBMS

The naming convention for a view is the same as for a table i.e., no space.

Some teams might have their own specific naming conventions for view, such as adding a vw\_ prefix.

Querying a VIEW is the same as querying a table:

When a view is queried, the query which is stored within the view is executed and the results are outputted as if they were part of a table

This means views can be treated like regular tables, e.g., you can join a regular table to the output of a view. However, unlike regular tables, as views do not contain data, we cannot modify the data in the view