Database Designs

Section 2: Database Tables

1.1 Introducing the SELECT Statement

Using the below Students Table we will look at how we can query the table to retrieve information from the database using SQL SELECT Statement.

STUDENTS				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.d@email.com
2	Jenny	Smith	F	j.s@email.com
3	Navdeep	Singh	F	n.s@email.com
4	Alberto	Diaz	M	a.d@email.com
5	Martha	Blyth	F	m.b@email.com

The SQL SELECT Statement is used to fetch data out of a database table(s). The **SELECT** statement is used to fetch specific rows and columns out of a database. Below is a very basic example of fetching data from a database using the **SELECT** Statement SQL Syntax.

We need to always answer three simple questions:
1. Which rows do we want to fetch?
2. Which columns in those rows are we interested in?
3. From which tables should we fetch this information from?

These are the three questions you need to think about in order to structure your thoughts and structure the **SELECT** statement.

If we want to get all emails of female students we can use the three questions to structure our thoughts and our **SELECT** statement to retrieve the data. First we are interested only in the rows which are female students. We are interested only in the column that holds the emails. Finally, we are interested in the Students Table which holds the information we want to extract. Having answered these questions we can now write the SQL query:

```
SELECT email
FROM STUDENTS
WHERE gender = 'F';
```

The keywords are highlighted in blue. These keywords are uniformly used to form the syntax of a basic **SELECT** statement. We would use these three keywords in a variety of ways to extract the most complicated of information from a database.
The **WHERE** keyword is used to answer the first question noted above i.e. which rows do we want to fetch. The **WHERE** clause is followed by a list of conditions which is used to figure out which of the rows we are interested in. Whichever rows satisfies the condition is selected as part of the select statement.
If we wanted all student emails from the database we can simply omit the **WHERE** clause from the **SELECT** statement.

The **SELECT** keyword is used to answer the second question noted above i.e. which columns are we interested in fetching. We select all the columns by the column names existing in the table separated by a comma. These are the values whose values will be retrieved for the records that satisfies the **WHERE** condition above.
The asterisk (*) is a special character to select all columns from a table rather than listing each column individually.

Finally, the **FROM** keyword is used to answer the last question noted above i.e. which table should we fetch the rows and columns data from. This should be a table stored in the database being queried.

The semicolon (;) at the end of the statement is extremely important because it terminates/ends the statement. This indicates to the DBMS system that we have completed our Query Statement.

Below is an example of a **SELECT** statement where we want to fetch all columns of all female students with the first name of Jenny.

```
SELECT *
FROM STUDENTS
WHERE gender = 'F' AND firstName = 'Jenny';
```

Below is an example of a **SELECT** statement where we want to fetch the student id and email of all female students or students with the first name of Alberto.

```
SELECT studentID, email
FROM STUDENTS
WHERE gender = 'F' OR firstName = 'Alberto';
```

The logical **AND/OR** operator is used to add multiple **WHERE** queries. We can chain the logical operators to produce more complex and interesting queries.

1.2 Columns Data Type

Database Columns have data types. There are different data types such as Strings, Numbers, Boolean, Null, etc. The data types for columns are specified when the tables

are created. The data types of columns govern how a column is treated in SQL queries. Below is a table of the various data types that can be created for databased tables:

Data Type	Description
Char	Holds fixed length strings.
Varchar	Holds variable length strings.
Int	Holds integer values i.e. full numbers.
Decimal	Holds floating point values i.e. decimal numbers.
DateTime	Holds the date and time stamp.
Date	Holds only the date stamp.
Time	Holds only the time stamp.
Blob	Holds binary larg objects. This holds data types that are not easily represented by the other data types.

1.3 Single Quotes, Escapes and NULLS

Data types of Char, Vachar, DateTime, Date and Time values all need to be enclosed in Single Quotes. What is the string itself contains a single quote? The escape character allows us to escape the single quote with a backslash (\ ).

```
SELECT buildingID
FROM property_address
WHERE buildingname = 'Akbar\'s House';
```

The escape characters tells the database that the character after the backslash should be taken literally and any special character recognition that character has is no longer true (i.e. accept the character literally as it).

Different database systems can accept different characters as the special escape characters. For example some databases escape a single quote with another preceding single quote. This will depend on the parsing rules of the database.

```
SELECT buildingID
FROM property_address
WHERE buildingname = 'Akbar''s House';
```

The NULL data value implies that a value does not exist. NULL is not the same as a blank string or the number zero. A Blank or a zero number is a value that exists. Any columns can contain a value of NULL and this can be defaulted to a column if no values are specified for the column for the data record.

To specify whether a column can have a value of NULL is done at the creation of the table. We have to define the table in a way that allows NULL values in that column.

The NULL values is neither True or False and is just a NULL/Non-existent value. To query a database for NULL values the syntax is slightly different.

```
SELECT studentID
FROM STUDENTS
WHERE email IS NULL;
```

```
SELECT studentID
FROM STUDENTS
WHERE email IS NOT NULL;
```

The equal logical operator (**=**) checks whether a value exists in a table. As mentioned above NULL is the absence of value. Therefore, we cannot use the equal logical operator to find a NULL value. Instead we would is **IS NULL** or **IS NOT NULL** to query whether a NULL value does or does not exist in the table.

1.4 Using the LIKE Operator

SQL allows us to use the **LIKE** keyword within the **WHERE** clause to retrieve data that contains a specific string within a string whether at the beginning, end or in-between the string. The below example demonstrates how to use the **LIKE** keyword to find 'gmail' within the string of the email column.

SELECT	Which Columns?	**email**
FROM	Which Tables?	**STUDENTS**
WHERE	Which Rows?	**email contains the string 'gmail'**

```
SELECT email
FROM STUDENTS
WHERE email LIKE '%gmail%';
```

The **LIKE** is a special keyword similar to **IS NULL** and **IS NOT NULL** special keywords but it allows us to use something called Wildcards. Wildcards are similar to wildcards in Regular Expressions and the symbols have special meanings. A wildcards can match portion of strings and the percentage symbol (**%**) is a wildcard.

The **%** wildcard means anything of any length i.e. this can be made up of characters, numbers, special characters, etc. In the above example the first **%** allows for anything before the 'gmail' string while the second **%** allows for anything after the 'gmail' string, even if that anything is nothing. Therefore, the **LIKE** keyword will match any string contain the string 'gmail' even if the string itself is 'gmail'.

The _ wildcard means anything of a length that is exactly one character i.e. whatever fills that underscore in the string position should be exactly one random character.

Wildcards are extremely useful; however, the make queries execute very slowly due to all the processing the database has to do in order to check and match against the wildcards.

1.5 BETWEEN, IN and NOT IN Operators

If we wanted to query whether a data is between two sets of numbers there are two ways in which we can write the query. The first method is using the math greater than and less than operators. The second method is to use the **BETWEEN** keyword. Below is an example of returning students who are between 20 and 25 years old.

```
SELECT studentID
FROM STUDENTS
WHERE age > 19 AND age < 26;
```

```
SELECT studentID
FROM STUDENTS
WHERE age BETWEEN 20 AND 25;
```

The **BETWEEN** operator is a very useful way of specifying a range which is much more easily readable and to write. The **BETWEEN** operator is inclusive and includes both the numbers at the beginning and end to the range.

Thus far we have only seen queries that queries against a single table. What if we we wanted to query for for some data across two tables. In the example below we have a Students and Subjects Tables where we are interested in returning subjectName of Students who are named ‘Jenny’ and ‘Navdeep’.

STUDENTS				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.d@email.com
2	Jenny	Smith	F	j.s@email.com
3	Navdeep	Singh	F	n.s@email.com
4	Alberto	Diaz	M	a.d@email.com

SUBJECTS		
studentID	subjectName	currentGrade
2	Mathematics	A
4	Chemistry	C

SUBJECTS		
1	English Language	D
3	History	NULL

```
SELECT subjectName
FROM STUDENTS, SUBJECTS
WHERE (firstName = 'Jenny ' OR firstName = 'Navdeep')
        AND STUDENTS.studentID =  SUBJECTS.studentID;
```

In the example above, in addition to querying for the firstName to be either 'Jenny' or 'Navdeep' from the Students table in our **WHERE** clause but we also need to connect the two tables together using a common value across both tables which happens to be the studentID column. An additional **WHERE** clause is added using the **AND** operator to query where the studentID from the STUDENTS table is equal to the studentID from the SUBJECTS table.

Only when both **WHERE** criteria's are satisfied only then can we retrieve the subjectName from the SUBJECTS table. We should pay special attention to the parenthesis between the two **WHERE** clauses because the parenthesis ensures the first condition is evaluated first before the second condition. This is similar to mathematics and the principal of BIDMAS where brackets are always evaluated first.

We can re-write the query above to make use of aliases instead of using the tables full name in order to make the query shorter and readable.

```
SELECT subjectName
FROM STUDENTS AS s, SUBJECTS AS c
WHERE (firstName = 'Jenny ' OR firstName = 'Navdeep')
        AND s.studentID =  c.studentID;
```

To create an alias we would use the **AS** keyword followed by the alias name within the **FROM** clause. We can name the alias anything we want and start using as a reference in our SQL statements **WHERE** clause.

Notice that we do not need to make a reference to the STUDENTS table for the firstName column. This is because firstName column is unambiguous as it appears only in the STUDENTS table. Similarly the subjectName column in the **SELECT** statement does not need reference to the SUBJECTS table because it is unambiguous. We would only need to make a reference the table name where the column appears in both tables and becomes ambiguous.

We can make the query above even more readable by using the **IN** keyword.

```
SELECT subjectName
FROM STUDENTS AS s, SUBJECTS AS c
WHERE firstName IN ('Jenny ', 'Navdeep')
        AND s.studentID =  c.studentID;
```

The **IN** operator is typically used when we have a set of values and we want to check whether any of those values within the list matches the column data. In the above example, if firstName matches either values within the list then the condition will evaluate to true. This removes the need to write very long **OR** clause lists. We can think of this like an Excel filter, filtering on a specific list of values.

The **NOT IN** operator is used to perform the opposite of the **IN** operator i.e. it checks whether the column data does not match the values within the list and excludes the specific list.

```
SELECT subjectName
FROM STUDENTS AS s, SUBJECTS AS c
WHERE (firstName <> 'Jenny ' OR firstName <> 'Navdeep')
        AND (s.studentID =  c.studentID);
```

```
SELECT subjectName
FROM STUDENTS AS s, SUBJECTS AS c
WHERE firstName NOT IN ('Jenny ', 'Navdeep')
        AND (s.studentID =  c.studentID);
```

The above two example will select students that are not 'Jenny' or 'Navdeep' and return their course subjects. We can use brackets to group each **WHERE** clauses to make the SQL statement more readable.

1.6 Multiple-Column SELECT

Not only can we select from multiple tables but we can also select multiple columns. We can demonstrate this by writing a SELECT statement that selects the subjectName and currentGrade columns for students whose surname begins with the letter S.

```
SELECT s.studentID, subjectName, currentGrade
FROM STUDENTS AS s, SUBJECTS AS c
WHERE (lastName LIKE 'S%')
        AND s.studentID =  c.studentID;
```

Matching by the studentID is very important in order to return the correct information about the correct student(s). Therefore, we must not forget to add in the second **WHERE** clause to match the two tables using the studentID columns.

Note that within the **SELECT** clause it does not matter which table we use to return the studentID data from. This is because we are using the **WHERE** clause to match the studentID and both tables will return the same results.

When matching the two tables using the studentID we can vision that it combines the two tables into one large table containing all table columns and data rows.

STUDENT/SUBJECTS						
studentID	firstName	lastName	gender	email	subjectName	current Grade
1	John	Doe	M	j.d@email.com	English Language	D
2	Jenny	Smith	F	j.s@email.com	Mathematics	A
3	Navdeep	Singh	F	n.s@email.com	History	NULL
4	Alberto	Diaz	M	a.d@email.com	Chemistry	C

Since we matched based on the StudentID we know that each row is the correct data for a single student. We can use this to vision to help us create our multi-column SELECT statements. For example, if we ant to retrieve the studentID, firstName, lastName, subjectName and currentGrade for students whose gender is female:

```
SELECT s.studentID, firstName, lastName, subjectName, currentGrade
FROM STUDENTS AS s, SUBJECTS AS c
WHERE (gender = 'F')
        AND s.studentID =  c.studentID;
```

We should always look out for ambiguity and ensure the columns selected or queried are specific and not ambiguous. By creating this large table we can filter out the column and rows data that we are interested in to return our final results. The results would look something like the below.

SELECT STATEMENT RESULTS				
studentID	firstName	lastName	subjectName	currentGrade
2	Jenny	Smith	Mathematics	A
3	Navdeep	Singh	History	NULL

1.7 Working with Dates & Times

Working with the Date and Time data types in databases is very common. The Date and Time functions vary between different DBMS and should be expected. Below are MySQL functions to demonstrate how we can work with Date and Time functions specific to that DBMS (*Always learn and understand how the Date and Time functions work with whichever DBMS you are working with*).

On a high level conceptually there are four common types of operations that you would use with dates: finding/manipulating the current date, splitting a date/time, creating dates and date arithmetics (e.g. find the difference between two dates).

Below are the functions we can use in MySQL for the four common type of operations:

<table>
<tr><th colspan="2">MYSQL DATETIME FUNCTIONS</th></tr>
<tr><th colspan="2">CURRENT DATE</th></tr>
<tr><td>SELECT NOW( )</td><td>Get the Current Date and Time (e.g. ‘2020-OCT-10 16:00:00’).</td></tr>
<tr><td>CURDATE( )</td><td>Get the Current Date (e.g. ‘2020-OCT-10’).</td></tr>
<tr><td>CURTIME( )</td><td>Get the Current Time (e.g. ’16:00:00’).</td></tr>
<tr><th colspan="2">SPLITTING DATES</th></tr>
<tr><td>EXTRACT(YEAR FROM Date)</td><td>Extracts the year from a date as an integer value.</td></tr>
<tr><td>EXTRACT(MONTH FROM Date)</td><td>Extracts the month from a date as an integer value.</td></tr>
<tr><td>EXTRACT(DAY FROM Date)</td><td>Extracts the day from a date as an integer value.</td></tr>
<tr><td colspan="2">Example:

SELECT EXTRACT(YEAR FROM date) AS rev_year,
EXTRACT(MONTH FROM date) AS rev_month,
EXTRACT(DAY FROM date) AS rev_day,
FROM sales_data
WHERE total_revenue = (SELECT MAX(total_revenue) FROM sales_data);

The MySQL EXTRACT command also allows us to parse out the week, quarter, hours and many other parts of the DateTime data type.</td></tr>
<tr><th colspan="2">CREATING DATES</th></tr>
<tr><td colspan="2">We can create dates from a string. All DBMS can support converting strings into dates and vice versa. For example we can convert a unambiguous and ambiguous dates (most DBMS are really good at converting strings into dates and vice versa):

SELECT * FROM sales_data WHERE date = “01-JAN-2020”;
SELECT * FROM sales_data WHERE date = “06-07-2020”;</td></tr>
<tr><td colspan="2">We can create dates from other dates using the DBMS functions. For example:

SELECT * FROM sales_data WHERE date = DATE_SUB(“2020-10-10”, INTERVAL 1 DAY);</td></tr>
<tr><th colspan="2">DATE ARITHMETICS</th></tr>
<tr><td>DATEDIFF( )</td><td>A function that takes in two dates and returns the number of days/ months/years (or whatever) between the two dates.

Example:

SELECT DATEDIFF(“2020-01-01”, “2020-02-01”) AS days_elapsed;</td></tr>
</table>

MYSQL DATETIME FUNCTIONS	
DATE_ADD( ) DATE_SUB( )	Functions that take a date and add/subtracts an interval from that date. The output of the function is a date. Example: **SELECT DATE_ADD(**“2020-01-01”, **INTERVAL** 1 **DAY) AS** date_tomorrow**;**

It is important to understand that the DATE and TIME functions vary a lot between DBMS to DMBS and you should ensure to understand the exact semantics of the functions for whichever DBMS you happen to be working with.