Database Designs

Section 2: Database Tables

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 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	currentGrad e
2	Jenny	Smith	Mathematics	A
3	Navdeep	Singh	History	NULL

2.7 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:

MYSQL DATETIME FUNCTIONS

CURRENT DATE

SELECT NOW( )	Get the Current Date and Time (e.g. ‘2020-OCT-10 16:00:00’).
CURDATE( )	Get the Current Date (e.g. ‘2020-OCT-10’).
CURTIME( )	Get the Current Time (e.g. ’16:00:00’).

SPLITTING DATES

EXTRACT(YEAR FROM Date)	Extracts the year from a date as an integer value.
EXTRACT(MONTH FROM Date)	Extracts the month from a date as an integer value.
EXTRACT(DAY FROM Date)	Extracts the day from a date as an integer value.

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.

CREATING DATES

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’;
```

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);
```

DATE ARITHMETICS

DATEDIFF( )	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;`

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.

2.8 Creating a Database, Use a Database and Create a Table

A Database (Abbreviated from Relational Database) is basically a collection of tables. There are no limit to the number of tables inside of a database. However, tables within a database are either implicitly or explicitly related.

There are three questions we would need to asks ourselves:
- How do we create databases?
- How do we create tables?
- How do we enter data into our tables?

These are the topics we will explore in the next few sections.

To create a database we would run a simple SQL statement within our DBMS where we pass in the name of the database we wish to create:

```
CREATE DATABASE ExampleDB;
```

The above statement will create a new database called ExampleDB which is the name of the database passed into the above command and could have called it anything we like.

Now that we have a database created we would need to run the following command to use the database so that all other commands (such as creating a table) will apply to the selected database.

```
USE ExampleDB;
```

The **USE** keyword will tell the DBMS to switch to the specified database and all SQL commands we execute will relate to this database.

Now that we have a database created and have switched to this database we can use the **CREATE TABLE** SQL command to create a table. Below is an example of creating a table called students.

```
CREATE TABLE Students (
	studentID INT NOT NULL AUTO_INCREMENT,
	firstName VARCHAR(30) NOT NULL,
	lastName VARCHAR(30) NOT NULL,
	gender CHAR(1),
	email VARCHAR(30) NOT NULL,
	PRIMARY KEY(student_id)
);
```

The above command will create the following empty table:

Students				
studentID	firstName	lastName	gender	email

To create a table we start with the **CREATE TABLE** command followed by the name of the table. The name of the table is what we would use across all SQL queries when we want to refer to that table.

Within the brackets we specify the names of all of the column that exists within the table. After each column name we need to specify certain column parameters in a particular order during the creation of the table. These parameters will govern how the columns within our table behave in the DBMS database.

The first thing we need to specify is the column's datatype. The datatype of the columns are specified at the time of table creation and these data types govern how a column is treated in SQL queries.

The next parameter we need to specify is whether NULL values are allowed for that particular column. Setting the value to **NOT NULL** specifies that NULL values are not allowed in the column while omitting **NOT NULL** allows NULL values.
Any column can contain a value of NULL provided that the table has been defined in a particular way that allows NULL values in that column.
Remember NULL implies that a value does not exists and is not a value in itself i.e. it indicates the absence of a value. Therefore, NULL is not the same as a blank string or zero values and is also neither true or false (i.e. these are all explicit values that exists).

A key is a set of columns whose values are unique for each row in a table. Therefore, a single column or set of columns can make up a key. A **PRIMARY KEY** is one such set of columns specified as the **PRIMARY KEY**. All tables must have a **PRIMARY KEY** and the database designer must designate one key as the **PRIMARY KEY** for the table. This is added at the very end using the **PRIMARY KEY** keyword followed by the column(s) name that make up this primary key.

How is a **PRIMARY KEY** different from any other key? DBMS will often construct an Index on the **PRIMARY KEY** automatically even without being told explicitly to do so. The DBMS will do something special with the **PRIMARY KEY** such that whenever we want to look up values using the **PRIMARY KEY** it will retrieve the data very quickly.

THIS GETS ASKED ON INTERVIEW QUESTIONS SO REMEMBER THIS!

The **PRIMARY KEY** can include either a single column or multiple columns and there no limit on the number of columns that can make up the **PRIMARY KEY** of a table. In the above example we only specified a single column within the round brackets as the **PRIMARY KEY**. We could have specified multiple columns and all those columns combined together would form the **PRIMARY KEY**.

Remember where multiple columns form a key, the values combined together form a unique value which identifies that particular row in the database. **PRIMARY KEY** columns can never contain NULL values ever.

A constraint or condition that the data in the table must satisfy and never violate and a **PRIMARY KEY** is one such type of constraint whereby the condition cannot be violated. NOT NULL is also a type of constraint. The database will throw an error whenever a constraint is violated and disallow the command that was trying to violating the constraint on the table. Therefore, we will never have data that violates the database table constraints if setup correctly i.e. we would always have valid data (data-integrity).

Finally, marking a column as **AUTO_INCREMENT** simply means that the database will keep track of inserting new values into this column for each insertion. The DBMS will keep track of all columns marked as **AUTO_INCREMENT** and will automatically take care of incrementing the value each time we insert a new record. Therefore, when we insert a new record we do not need to specify a value for a column marked with the **AUTO_INCREMENT** parameter keyword. Therefore, we can always explicitly specify a value for the column in any case if we would like to or leave the value out and the database will take care of inserting a valid value for that column.
(In MS-SQL Server, the keyword **INDEX** is used instead of **AUTO_INCREMENT**).

2.9 Insert Table Further Example

Below are some more examples of **CREATE TABLE** statements with different column data types and multi-column Primary Key:

```
CREATE TABLE Sales_Data (
        storeLocation VARCHAR(30) NOT NULL,
        productName VARCHAR(30) NOT NULL,
        saleDate DATE NOT NULL,
        revenue DEC(10, 2) NOT NULL DEFAULT 0.0,
        PRIMARY KEY(storeLocation, productName, saleDate)
);
```

Sales_Data			
storeLocation	productName	salesDate	revenue

It is important to note that the table name should not contain any spaces. We can use camelCase, PascalCase, snake_case or kebab-case for naming conventions - in the above we are using the snake_case naming convention.

The **DATE** datatype indicates that the column should hold date values. We can insert a date using strings (enclosed in single/double quotes - single quotes are more widely accepted by DBMS. It is good practice to use single quotes). Dates can be accepted in a whole variety of formats and will accept the date and internally store it in the system format. There are different date formats we can use below are some examples:

‘11-Oct-2020’, ‘October-11-2020’, ‘10-11-2020’, ’10/11/2020’, etc.

It is important to note that dates are formatted using US format by default and therefore when adding dates using GB format we should be careful when using date values consisting numbers only.

The **DEC** datatype indicates that the column holds decimal values. The first value within the **DEC** parameter dictates the number of upper digits before the decimal point and the second value dictates the number of digits after the decimal point. Any monetary value in a table should have the datatype of **DEC**.

The **DEFAULT** key allows us to set a default value constraint so that the DBMS knows what to do if the value does not exist i.e. if an insert statement skips the value of this column the default value will be assigned instead of NULL.

Finally, the multi-column **PRIMARY KEY** indicates that the column storeLocation alone is not unique, productName alone is not unique and saleDate alone is not unique; however, the three column combined together would make a unique value.
Therefore, the combination of these three columns must be unique and no two rows within this table can have the same multi-column **PRIMARY KEY** value. Such keys are called Multi-Attributed, Multi-Column or Composite keys.

2.10 Insert Table Data

The SQL **INSERT** statement is used to insert data into a table. Below is an example syntax for inserting data:

```
INSERT INTO TABLE Students (firstName, lastName, gender, email)
VALUES (‘John’, ‘Doe’, ‘M’, ‘j.doe@email.com’);
```

We use the **INSERT INTO TABLE** command followed by the name of the table we want to insert new data into. Within the brackets we list the column(s) we wish to provide data values for. The **VALUES** keyword is then used to provide the data values within brackets for each column specified in the **INSERT** statement. The ordering of values must correlate to the column(s) listed i.e. there must be a 1:1 match between the column name and it’s inserted value. When we insert the row of data into the table we will notice that the studentID column will be auto-populated by the DBMS using the **AUTO_INCREMENT** even though we did not specify this column and value in the **INSERT** statement. The will **AUTO_INCREMENT** this number whenever a new data is inserted into this table. The table will now look like something below.

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com

(Important Note: we could have provided a unique value for the studentID on insertion but if omitted this from the **INSERT** statement the DBMS will auto-populate this value using **AUTO_INCREMENT**. The studentID will never be NULL and will always satisfy the **NOT NULL** constraint).

We can specify the table columns out-of-order in the **INSERT** statement and it does not need to be in the same order as found in the table itself. Below is an example of the above statement but where the columns specified in the **INSERT** statement are out-of-order from the table. Remember the values for insertion must correlate to the order of the columns specified in the **INSERT** statement.

```
INSERT INTO TABLE Students (gender, lastName, email, firstName)
VALUES (‘F’, ‘Smith’, ‘j.smith@email.com’, ‘Jenny’);
```

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	jess.doe@email.com

The name of the columns within the **INSERT** statement is used to perform the mapping of the inserted data into the correct table columns.

The **INSERT** statement syntax can be written in short-form whereby we do not explicitly specify the table columns and instead only provide the values. Using this short-form syntax we must ensure the values are in the order of the columns as they appear in the table itself. The DBMS will implicitly map the inserted values in the order of the columns as found in the database table.

```
INSERT INTO TABLE Students
VALUES (‘Navdeep’, ‘Singh’, ‘F’, ‘n.singh@email.com’);
```

The studentID can be ignored in the inserted values because the DBMS will auto-populate this value for us. We need to ensure we list the values for all the column that requires us to provide a value for.

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com

Finally, the below example demonstrates the ability to provide a value for the studentID if we wish to do so when we insert new data into the table. It is important that we provide a unique value for the studentID otherwise the DBMS will error when it tries to perform the **INSERT** command.

INSERT INTO TABLE Students
VALUES (‘4’, ‘Alberto’, ‘Diaz’, ‘M’, ‘a.diaz@email.com’);

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com

If we have a column in our table which accepts NULL values then its value can be skipped in the **INSERT** statement. Therefore, using the first syntax example we do not need to explicitly list the column whose value will be NULL.

INSERT INTO TABLE Students (email, firstName, lastName)
VALUES (‘jess.doe@email.com’, ‘Jesica’, ‘Doe’);

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com
5	Jessica	Doe	NULL	jess.doe@email.com

Where we omit a value for a column that accepts NULL values, the DBMS will automatically assign NULL indicating the absence of data/value in that column.

The **AUTO_INCREMENT** property increments the value of a particular column by 1 for every insertion. What would happen if we inserted a new data into the table but specified the studentID value that did not follow the natural sequence? DBMS will handle this automatically and will take care of out-of-sequence additions to the table which has **AUTO_INCREMENT** set on.

```
INSERT INTO TABLE Students
VALUES ('99', 'Carl', 'Flint', 'M', 'c.flint@email.com');
```

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com
5	Jessica	Doe	NULL	jess.doe@email.com
99	Carl	Flint	M	c.flint@email.com

```
INSERT INTO TABLE Students
VALUES ('Diane', 'Morris', 'F', 'd.morris@email.com');
```

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com
5	Jessica	Doe	NULL	jess.doe@email.com
99	Carl	Flint	M	c.flint@email.com
100	Diane	Morris	F	d.morris@email.com

The DBMS will keep track of the highest number within the table and will **AUTO_INCREMENT** by 1 from that number as seen in the above example.

2.11 Referential Integrity (Foreign Key Constraints)

Below are two tables, Students table and Campus_Housing tables. The Campus_Housing table contains the dormitory that the student lives at. These two tables are linked by the studentID.

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com

Campus_Housing		
studentID	dormitoryName	apartmentNumber
1	Brasenose Hall	115
2	Keble House	254
3	Pembroke House	331
4	NULL	NULL

What exactly is the link between these two tables?

We know that the Students table contains one row per student i.e. the row contains information for one student while the Campus_Housing table contains one row for the housing information of each student.

The Campus_Housing table should only contain rows for students who exists in the students table. Wy is this? The Campus_Housing table only makes sense when viewed in relation to the students table i.e. does not make sense when looking at it by itself. The entry in this table only makes sense if there is a corresponding matching entry in the students table which indicates that the student exists and attends this particular school.

The Students table will exist even if the Campus_Housing table does not, but the reverse is not true. For example, you can have a student that exists but does not live at a campus accommodation because they live off-campus with their parents. Therefore, the student can validly exists in the Students table with no entry in the Campus_Housing table.

Therefore, each value in the studentsID column of the Campus_Housing must already exists in the StudentID column of the Students table.

We can add new rows to the Student table without adding them to the Campus_Housing table. Therefore we can have tables like the below which will be totally valid:

Students				
studentID	firstName	lastName	gender	email
1	John	Doe	M	j.doe@email.com
2	Jenny	Smith	F	j.smith@email.com
3	Navdeep	Singh	F	n.singh@email.com
4	Alberto	Diaz	M	a.diaz@email.com
5	Jessica	Doe	NULL	jess.doe@email.com
99	Carl	Flint	M	c.flint@email.com
100	Diane	Morris	F	d.morris@email.com

Campus_Housing		
studentID	dormitoryName	apartmentNumber
1	Brasenose Hall	115
2	Keble House	254
3	Pembroke House	331
4	NULL	NULL

Jessica, Carl and Diane are valid existing students but they do not live at a campus accommodation i.e. they live at private accommodations. The contents of the two tables above continue to make common sense.

The above logic needs to be explicitly specified in the table specification of the Campus_Housing table. Therefore when we use the **CREATE TABLE** command to setup the Campus_Housing table we need to specify some additional parameters in order to ensure that every studentID that is entered into this table also has a corresponding entry in the Students table. The syntax can be seen below:

```
CREATE TABLE Campus_Housing (
        studentID INT NOT NULL,
        dormitoryName VARCHAR(50),
        apartmentNumber INT,
        CONSTRAINT fk_student_studentid,
        FOREIGN KEY(studentID),
        REFERENCES Students(studentID)
);
```

The **CONSTRAINT** keyword creates a constraint on the data that must be satisfied by the Campus_Housing table. The constraint above tells the Campus_Housing table that each value in the studentID column of the Campus_Housing table must already exist is the studentID column of the Students table. Such constraints are called **FOREIGN KEY** Constraints or Referential Integrity Constraints.

The term **FOREIGN KEY** is used because the constraint relates to a column that is a key in another table.

The term Referential Integrity is used because one table refers to another table to check the integrity of its own data. The Students.studentID is a key in the Students table and it is referenced by the Campus_Housing.studentID to check if it's integrity is maintained. The integrity is maintained when this condition returns true.

The last three lines of the **CREATE TABLE** SQL command tells the database of this constraint:

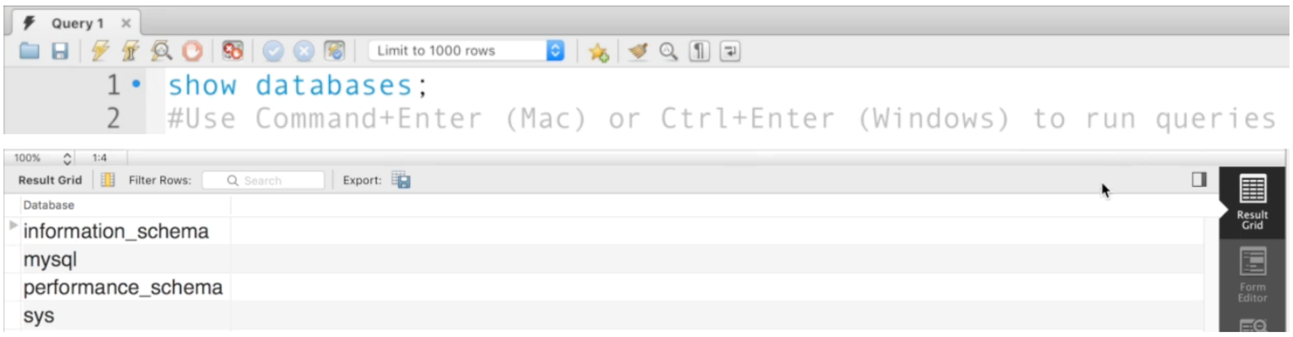
The first line **CONSTRAINT** fk_student_studentid, tells the database to create a constraint called fk_student_studentid. We could have named this constraint anything we like although it is best practice to use meaningful names.

The second line **FOREIGN KEY(**studentID**)**, tells the database to create a Foreign Key using the studentID column from the Campus_Housing table i.e. a column from the created table i.e. Campus_Housing.studentID.

Finally, the last line **REFERENCES** Students**(**studentID**)** tells the database that the Foreign Key references the studentID from the Students table i.e. Students.studentID.

2.12 Creating a Database and Using it in MySQL

Below is an example of creating a database and using it in MySQL. Remember the SQL commands can vary from database to database and you should always look at the documentation to use the correct syntax for the database being used. Below are screenshots from MySQL using the MySQL Workbench GUI.



```
show databases;
#Use Command+Enter (Mac) or Ctrl+Enter (Windows) to run queries
```

```
Database
information_schema
mysql
performance_schema
sys
```

Important Note: there are a few system meta databases that come pre-loaded in MySQL databases which are information_schema, mysql, performance_schema and sys. You should not delete or change these databases.

Below are some other example commands creating a National Stock Exchange database:

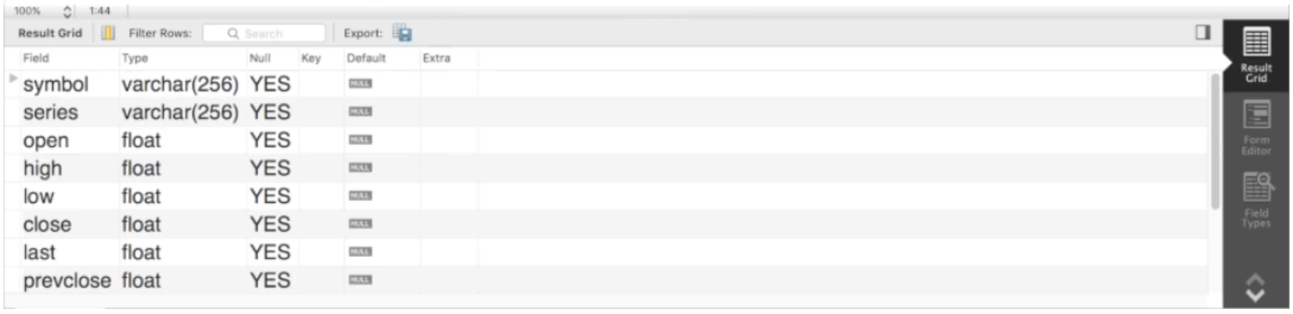
```
CREATE database nse;
use nse;
show tables;
```

```
create table stockMovements(
symbol varchar(256),
series varchar(256),
open float,
high float,
low float,
close float,
last float,
prevclose float,
tottrdqty float,
tottrdval float,
timestamp varchar(256),
totaltrades float,
isin varchar(256)
);
```

Using Varchar datatype would allow the column to have a string data unto 256 characters. The Char datatype would always result in a data of 256 characters and a string with less than 256 characters would be padded with empty blanks characters to make up the difference. We should always use the Varchar datatypes for storing strings data if we are uncertain about the length of the string.

There are some columns that are MySQL keywords such as open, close, last, etc. which are highlighted in blue. This does not prevent us from naming our columns after these keywords.

```
desc stockMovements;
```



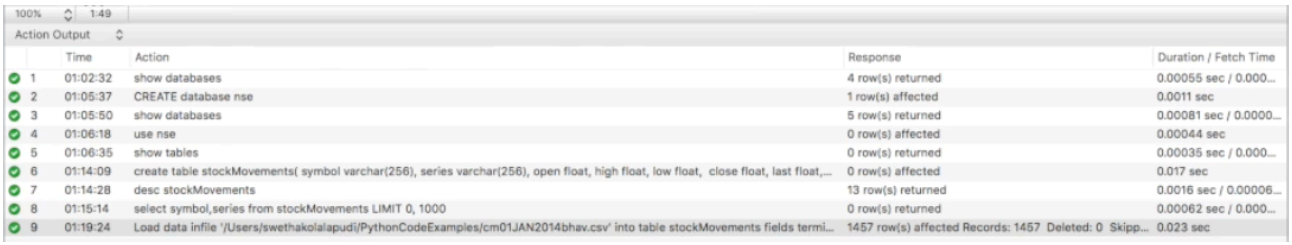
Field	Type	Null	Key	Default	Extra
symbol	varchar(256)	YES		NULL	
series	varchar(256)	YES		NULL	
open	float	YES		NULL	
high	float	YES		NULL	
low	float	YES		NULL	
close	float	YES		NULL	
last	float	YES		NULL	
prevclose	float	YES		NULL	

The **DESC** command allows us to see the description of the table i.e. the columns, datatype and constraints.

```
Load data infile ''
into table stockMovements fields terminated by ',' ignore 1 lines;
```

MySQL has a **LOAD DATA** command. The **LOAD DATA** command is followed by the path of the file we wish to load into our MySQL database. The **INFILE** keyword specifies that the loaded files is an input file. The **FIELDS TERMINATED** keyword specifies the delimiter in the file while the **IGNORE** keyword tells the database how many lines to skip/ignore from the loaded file.

Bulk loading files into a database tables is a common operation that are carried out in real databases. Behind the scenes the DBMS knows how to parse a file and then it runs a **INSERT** statement for each line of the file using the information extracted from the file. Important Note: when bulk loading a file we should be careful to correctly specify the field delimiter (MySQL default is tab ('\t')) and to eliminate file headers.



100% 1:49

Action Output

		Time	Action	Response	Duration / Fetch Time
✓	1	01:02:32	show databases	4 row(s) returned	0.00055 sec / 0.000...
✓	2	01:05:37	CREATE database nse	1 row(s) affected	0.0011 sec
✓	3	01:05:50	show databases	5 row(s) returned	0.00081 sec / 0.0000...
✓	4	01:06:18	use nse	0 row(s) affected	0.00044 sec
✓	5	01:06:35	show tables	0 row(s) returned	0.00035 sec / 0.000...
✓	6	01:14:09	create table stockMovements(symbol varchar(256), series varchar(256), open float, high float, low float, close float, last float,...	0 row(s) affected	0.017 sec
✓	7	01:14:28	desc stockMovements	13 row(s) returned	0.0016 sec / 0.00006...
✓	8	01:15:14	select symbol,series from stockMovements LIMIT 0, 1000	0 row(s) returned	0.00062 sec / 0.000...
✓	9	01:19:24	Load data infile '/Users/swethakolalapudi/PythonCodeExamples/cm01JAN2014bhav.csv' into table stockMovements fields termi...	1457 row(s) affected Records: 1457 Deleted: 0 Skipp...	0.023 sec

The MySQL Action Output Log window will provide us the time, statement command and response for every command that we have run. This is useful to see whether a command ran successfully (such as the above command for bulk loading).

Now that the table has some data, the **SELECT** statement can be used to query this data for example:

- selecting all table columns from the stockMovements table and limiting to the first 10 data, or
- return the number of data from the stockMovements table, or
- return the number of unique/distinct items in the symbol column from the stockMovements and setting the return column with an alias, or
- return the min, max and average opening price from the stockMovements table and setting the return column with alias's, or
- return the stock which had highest opening price using the **FROM** clause

```
select * from stockMovements limit 10;
select count(*) from stockMovements;
select count(distinct symbol) as numTickers from stockMovements;
select max(open) as maxOpen, min(open) as minOpen, avg(open) as
avgOpen from stockMovements;
select symbol, series from stockMovements where open=19399;
```

2.13 Advanced Bulk Loading in MySQL

In the previous section we saw a very simple way of loading in data into a table that had no constraints. We will now look at a more advanced syntax for bulk loading data into MySQL including columns that contain constraints.

```
create table tickers(
id int(11) unsigned not null auto_increment,
symbol varchar(256),
series varchar(256),
isEquity int,
primary key(id),
unique key (symbol,series)
);
```

As we can see in this create table example, it uses the features we have already seen for creating tables in the previous sections such as **NOT NULL**, **AUTO_INCREMENT**, **PRIMARY KEY**, etc. The **UNIQUE KEY** keyword constraint is used to create a multi-column candidate key.

Loading data into this table is a lot more complicated and the load command is equally complicated.

```
Load data infile ' '
ignore INTO TABLE ticker
fields terminated by ',' IGNORE 1 lines
(symbol,@series,@open,@high,@low,@close,@last,
@prevclose,@tottrdqty,@tottrdval,@timestamp,@totaltrades,@isin)
SET series=@series,isEquity=if(@series='EQ',1,0)
```

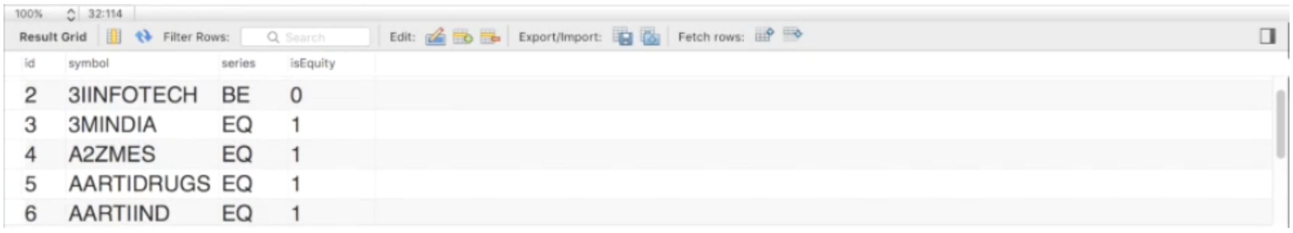
The keyword **IGNORE INTO TABLE** tells the database that if there is a row which for any reason cannot fit into the ticker's table (and its constraints) then it should ignore loading that row of data - this will skip loading in any duplicate data as there is a unique multi-column constraint on the symbol and series column.

The **TERMINATE BY** specifies the delimiter for the loaded in file as well as to ignore the header line.

The syntax contained within the round brackets is the more interesting and advanced part of this load command. The values in the brackets corresponds to the list of columns in the loader file. This lists 1:1 of the column in the same order as found in the loader file. Values preceded with a @ symbol are assigned to variables. We can use these variables to explicitly **SET** column values as seen with the series and isEquirty column or do nothing at all with them in which case those column in the loader file will be ignored.

In the above we check whether the value of @series is equal to 'EQ' and if it is 1 true then 1 is assigned to the column isEquity else if false the value is assigned 0 in the isEquity column. If we run this command with a CSV loader file, we would see something like the below:

```
select * from tickers limit 10;
```



id	symbol	series	isEquity
2	3IINFOTECH	BE	0
3	3MINDIA	EQ	1
4	A2ZMES	EQ	1
5	AARTIDRUGS	EQ	1
6	AARTIIND	EQ	1

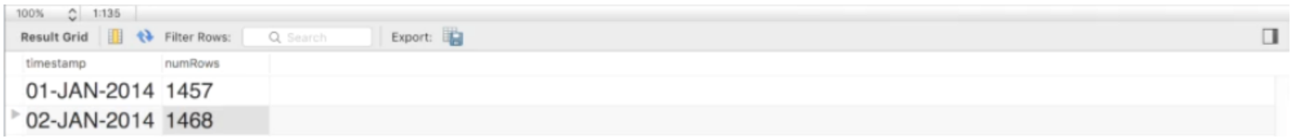
Notice that where the series column has a value other than 'EQ' the isEquity column value is set to 0. Also the id column is incremented by 1 due to the **AUTO_INCREMENT** constraint on the id column.

If we were to load in a second loader file into the two tables (stockMovements and tickers) we should notice that non-duplicate values will only load into the tickers table due to that table's constraints.

```
Load data infile ' '
into table stockMovements fields terminated by ',' ignore 1 lines;
```

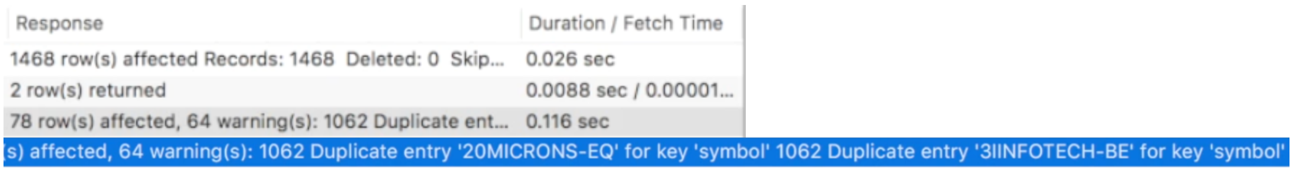
The below command displays the number of rows for each of the two files loaded grouped by the timestamps.

```
select timestamp, count(symbol) as numRows from
stockMovements group by timestamp;
```



```
timestamp    numRows
01-JAN-2014  1457
02-JAN-2014  1468
```

If we were to look at our output action log we should notice a detailed response for the second load executed load command similar to the below. This will display the total rows inserted and the total rows rejected/ignored.



```
Response                                                   Duration / Fetch Time
1468 row(s) affected Records: 1468  Deleted: 0  Skip...    0.026 sec
2 row(s) returned                                          0.0088 sec / 0.00001...
78 row(s) affected, 64 warning(s): 1062 Duplicate ent...   0.116 sec
s) affected, 64 warning(s): 1062 Duplicate entry '20MICRONS-EQ' for key 'symbol' 1062 Duplicate entry '3IINFOTECH-BE' for key 'symbol'
```

This is a more typical example of bulk loading where the command is more advanced and takes into account the constraints of a table.

END