Beginning SQL

"Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation) … activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed."

The architecture of any new application that is being built depends on the responsibility that it takes on ***handling the Data***. And the reason is pretty obvious as data is nothing but information, and to handle this information most of the applications are being built today.

As the technologies are evolving and as applications/packages are being developed to handle a specific unit in a big task, the architecture of any new application (discussed earlier) depends on to what extent it should handle the data and leave the rest to the ***package*** that is exclusively designed for *Data Handling/Data Management.*

***R**elational **D**ata**B**ase **M**anagement **S**ystem (RDBMS)*

"A relational database is a big spreadsheet that several people can update simultaneously."
Each *table* in the database is one spreadsheet. You tell the RDBMS how many columns each row has.
For example, in a mailing list (where you need to store email addresses of persons) database, the table has two columns: `name` and `email`. In this case, each entry in the database consists of one row and two columns in this table.
An RDBMS is more **restrictive** than a spreadsheet in that all the data in one column must be of the same type, e.g., integer, decimal, character string, or date.

What is SQL?

SQL (pronounced "ess-que-el") stands for Structured Query Language.
SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Microsoft SQL Server, Oracle, Sybase, Access, Ingres, etc.

Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

Table Basics

A relational database system contains one or more objects called tables. The data or information for the database is stored in these tables. Tables are uniquely identified by their names and are comprised of columns and rows. Columns contain the column name, data type, and any other attributes for the column. Rows contain the records or data for the columns.

Here is a sample table called "weather".
City, state, high, and low are the columns. The rows contain the data for this table:

Weather			
city	**state**	**high**	**low**
Phoenix	Arizona	105	90
Tucson	Arizona	101	92
Flagstaff	Arizona	88	69
San Diego	California	77	60
Albuquerque	New Mexico	80	72

The SELECT Statement

We use the SELECT statement to select (choose, extract) data from database columns. The SELECT statement is used in conjunction with another statement called the **FROM** statement.
The FROM statement is used to identify the name of the table in which the columns of data we are interested in are stored.

The SELECT statement has five main clauses to choose from, although, FROM is the only required clause. Each of the clauses have a vast selection of options, parameters, etc. The clauses will be listed below, but each of them will be covered in more detail later in the material.

The column names that follow the select keyword determine which columns will be returned in the results. You can select as many column names that you'd like, or you can use a "*" to select all columns.

The table name that follows the keyword **from** specifies the table that will be queried to retrieve the desired results.

Here is the format of the SELECT statement:

```
SELECT [ALL | DISTINCT] column1[,column2]
FROM table1[,table2]
[WHERE "conditions"]
[GROUP BY "column-list"]
[HAVING "conditions]
[ORDER BY "column-list" [ASC | DESC] ]
```

SELECT All Columns

```
SELECT *
FROM table_name
```
MEANING: it means to select all (* means all) columns in the table.

Example

```
SELECT *
FROM Weather
```

SELECT Specific Columns

```
SELECT column_name(s)
FROM table_name
```
MEANING: it means to select certain (named) columns from the table.

Example
```
SELECT City, State
FROM Weather
```

SELECT a Certain Number of Rows

```
SELECT Top < number of rows >
FROM table_name
```
MEANING: it means to select a certain number of rows from all columns in the table

Example
```
SELECT Top 2
FROM Weather
```

SELECT a Percentage Number of Rows

```
SELECT Top < percentage of rows >
FROM table_name
```
MEANING: it means to select a percentage number of rows from all columns in the table.

Example
```
SELECT Top 33 percent
FROM Weather
```

Position of Semicolon in SQL Statements?

Semicolon is a standard way to end an SQL statement (and to separate different SQL statements) in systems that allow more than one SQL statement to be executed in the same call to a database server, but if you don't use these programs it **not** necessary to end the SQL statement by semicolon.

The **ALL** and **DISTINCT** Statement

ALL and **DISTINCT** are keywords used to select either ALL (default) or the "distinct" or unique records in your query results. If you would like to retrieve just the unique records in specified columns, you can use the "DISTINCT" keyword. DISTINCT will discard the duplicate records for the columns you specified after the "SELECT" statement

ALL will display "all" of the specified columns including all of the duplicates. The ALL keyword is the default if nothing is specified.

SYNTAX

```
SELECT DISTINCT column_names
FROM table_name
```

MEANING: DISTINCT keyword used to return only distinct (different) values in the column.

Examples

```
SELECT DISTINCT State
FROM Weather

SELECT DISTINCT City
FROM Weather
```

The **WHERE** statement

The WHERE *clause* is used to *filter* search results and can be assigned many *filter conditions* or *clause operators*.
In this section we will study the simplest *clause operators* which are used to compare data held in a database column with a filter condition (search criteria) applied by you.

Operator	Description
=	Equal To
>	Greater Than
<	Less Than
<>	Non-Equality (Not Equal To)
!=	Non-Equality (Not Equal To)
!<	Not Less Than
!>	Not Greater Than
>=	Greater Than Or Equal To
<=	Less Than Or Equal To
BETWEEN	Between two specified values
IS NULL	Is a NULL value
Note that some operators perform identical tasks. For example *<> and !=* can both represent *Not Equal To*. By the same token, *!<* (*not less than*) has the same effect as *>=* (*greater than or equal to*). Different database systems may prefer using different operators. That's why we have shown them all here.	

And also…

Operator	Description
LIKE	Looks for a similar pattern within the data
AND	Logical AND (i.e. green *AND* big)
OR	Logical OR (i.e. green *OR* big)

SYNTAX

```
SELECT column_name
```

```
FROM table_name
WHERE <operator> <operator value>;
```

EXAMPLES

1. ```
   SELECT *
   FROM Weather
   WHERE low = 90;
   ```
   Meaning: SELECT all records FROM the Weather table WHERE their low is equal to 27

2. ```
   SELECT *
   FROM Weather
   WHERE low > 75;
   ```
   Meaning: SELECT all records FROM the Weather table WHERE their low is greater than 27

3. ```
   SELECT *
   FROM Weather
   WHERE low >= 60;
   ```
   Meaning: SELECT all records FROM the Weather table WHERE their low is greater than or equal to 27

4. ```
   SELECT City, State, High
   FROM Weather
   WHERE low !=27;
   ```
   Meaning: SELECT only the City, State, High from theWeather table WHERE their low is not equal to 27

We must use single quotes when dealing with text, like so 'text' . the rule is, if it's a ***integer*** (number that can have calculations performed on it) we do not use quotes. If it is a ***string*** (either words or numbers that do not have calculations applied to them) we use quotes. So if your database has a text field with the number *87* or the word *Artist* in it we use quotes '87' and 'Artist'. If the number *87* is held on the database in a numerical field we do not use the quotes.

1. ```
   SELECT *
   FROM Weather
   WHERE City = 'Tucson'
   ```
   Meaning: SELECT all records FROM the Weather table WHERE city is Tucson.

2. ```
   SELECT *
   FROM Weather
   WHERE City > 'FlagStaff'
   ```
   Meaning: SELECT all records FROM the Weather table WHERE City is alphabetically greater than FlagStaff

3. ```
   SELECT *
   FROM Weather
   WHERE City != 'FlagStaff'
   ```
   Meaning: SELECT all records FROM the Weather table WHERE City is NOT FlagStaff

4. ```
   SELECT *
   FROM Weather
   WHERE City <= 'San Diego'
   ```
   Meaning: SELECT all records FROM the Weather table WHERE City is alphabetically less than or equal to San Diego

GROUP BY clause

The GROUP BY clause will gather all of the rows together that contain data in the specified column(s) and will allow aggregate functions to be performed on the one or more columns. This can best be explained by an example:

SYNTAX

```
SELECT column1,
SUM(column2)
FROM "list-of-tables"
GROUP BY "column-list";
```

Employee				
FirstName	**LastName**	**Department**	**Age**	**Salary**
Guru	Raj	IT	28	2200
Rohit	Bal	FT	36	3600
Neetu	Lulla	FT	19	2600
Paul	Phoenix	ARMY	90	30000
Steve	Fox	ARMY	25	32500

Let's say you would like to retrieve a list of the highest paid salaries in each dept:

```
SELECT Max(Salary), Department
FROM Employee
GROUP BY Dept;
```

This statement will select the maximum salary for the people in each unique department. Basically, the salary for the employee who makes the most in each department will be displayed. Their salary and their department will be returned.

What if we want to display their lastname too?

Modify the above query to..

```
SELECT Max(Salary), Department, LastName
FROM Employee
GROUP BY Department, LastName;
```

This is a called "Multiple Grouping Columns".

HAVING clause

The HAVING clause allows you to specify conditions on the rows for each group - in other words, which rows should be selected will be based on the conditions you specify. The HAVING clause should follow the GROUP BY clause if you are going to use it.

SYNTAX

```
SELECT column1,
SUM(column2)
FROM "list-of-tables"
GROUP BY "column-list"
HAVING "condition";
```

HAVING can best be described by example. Let's say you have an employee table containing the employee's name, department, salary, and age. If you would like to select the average salary for each employee in each department, you could enter:

```
SELECT Department, Avg(salary)
FROM Employee
GROUP BY Department;
```

But, let's say that you want to ONLY calculate & display the average if their salary is over 20000:

```
SELECT Department, Avg(Salary)
FROM Employee
GROUP BY Department
HAVING Avg(Salary) > 20000;
```

ORDER BY Clause

ORDER BY is an optional clause which will allow you to display the results of your query in a sorted order (either ascending order or descending order) based on the columns that you specify to order by.

SYNTAX

```
SELECT column_name
FROM  table_name
ORDER BY column_name
```

MEANING: it means to sort the rows of the named column in the table.
There are two clauses for the ORDER BY statement: ASC & DESC

Sort in Ascending order

To sort the rows in alphabetical order or numbers in numerical order from lowest to highest.
Note: The ascending clause is the default clause in the order statement. So if you do not specify ASC or DESC the results will automatically be sorted in ascending order.

EXAMPLE

```
SELECT FirstName, Department, Age
FROM Employee
ORDER BY Age
```

Sort in Descending Order

To sort the rows in reverse alphabetical order and the order numbers in numerical order from high to low.

SYNTAX

1.
```
SELECT column_name
FROM  table_name
ORDER BY column_name DESC;
```

2.
```
SELECT  FirstName, Department, Age , Salary
FROM Employee
ORDER BY Department, Salary, FirstName DESC
```

How to order multi columns

First we have to decide which column has priority over the others when it comes to sorting. In the example below "FirstName" gets sorted first (our 1st priority) after which age (our 2nd priority) gets sorted *within the results* produced by our first priority sort.

EXAMPLE

1.
```
SELECT  FirstName, Department, Age
FROM Employee
```

2.
```
SELECT  FirstName, Department, Age ,Salary
FROM Employee
```

3.
```
SELECT  FirstName, Department, Age , Salary
FROM Employee
ORDER BY Department, Salary, FirstName
```

Note: The sorting of the records will happen in the Order you specify in your Query

Mixed Ascending and Descending in order by statement.

EXAMPLES

a.
```
SELECT  FirstName, Department, Salary
FROM Employee
ORDER BY Department ASC, Salary DESC;
```

b.
```
SELECT  FirstName, Department, Salary
FROM Employee
ORDER BY Department DESC, Salary ASC
```

LIKE Clause

The **LIKE** pattern matching operator can also be used in the conditional selection of the where clause. Like is a very powerful operator that allows you to select only rows that are "like" what you specify. The percent sign "%" can be used as a wild card to match any possible character that might appear before or after the characters specified.
SYNTAX

```
SELECT column_name
FROM table name
WHERE column_name LIKE 'matching clause'
```

Meaning: *WHERE* the column_name is something *LIKE* the contents of whatever keyword we are using in the *'matching clause'* . When using LIKE we can widen our choices by using a wildcard symbol *%* an underscore _ , or both.

Wildcard character	Description	Example
%	Any string of zero or more characters.	WHERE title LIKE '%dad%' finds all book titles with the word 'dad' anywhere in the book title.
_ (underscore)	Any single character.	WHERE title LIKE '*_ad*' finds all *three letter* words that end with *ad* (dad, bad, mad and so on).

Examples

1.
```
SELECT *
FROM Employee
WHERE FirstName LIKE '_aul';
```
Meaning: SELECT all records FROM the Employee table WHERE the first name consists of 4 letters, the last 3 of which are *'aul'*

2.
```
SELECT *
FROM Employee
WHERE FirstName LIKE 'N%';
```
Meaning: SELECT all records FROM the Employee table WHERE the first name starts with the letter N

The wildcard % ensures that all names beginning with N are displayed no matter what the rest of the letters are.

3.
```
SELECT *
FROM Employee
WHERE Department LIKE '%r';
```
Meaning: SELECT all records FROM the Employee table WHERE the *last* letter of the Department is *r*

4.
```
SELECT *
FROM Employee
WHERE Department LIKE '%er';
```

Meaning: SELECT all records FROM the Employee table WHERE the last two letters of the Department title are e*r.*

5.
```
SELECT *
FROM Employee
WHERE FirstName LIKE '%me%';
```
Meaning: SELECT all records FROM the Employee table WHERE the group of letters ***me*** appear *anywhere* within the first name.

6.
```
SELECT *
FROM Employee
WHERE FirstName LIKE '%e_' AND Age > 75
ORDER BY Age DESC;
```
Meaning: SELECT all records FROM the Employee table WHERE the the letter e is the second to last letter in the first name and the candidates Age is greater than 75. Order the results by Age in Descending order.

BETWEEN…. AND Clause

The BETWEEN conditional operator is used to test to see whether or not a value (stated before the keyword BETWEEN) is "between" the two values stated after the keyword BETWEEN.

SYNTAX

```
SELECT column_name
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Meaning: Using BETWEEN …. AND will create a search that returns values between ***value1*** and ***value2*** thus presenting us with a range of data placed between the two values we are interested in. Values can be numbers, text, or dates.

As with most things, there are similar ways of achieving the same result as we get using BETWEEN. Here are some examples ....

SQL Syntax

WHERE column_name BETWEEN ***value1*** AND ***value2***;

is equivalent to
WHERE column_name ***> value1*** AND column_name ***< value2***;
Note that BETWEEN ... AND may return different results depending on what database is being used. Some of these differences are shown below ....

WHERE column_name BETWEEN ***value1*** AND ***value2***;
could be equivalent to

WHERE column_name ***> value1*** AND column_name ***< value2***;

or Equivalent to
WHERE column_name ***>= value1*** AND column_name ***< value2***;

or Equivalent to
WHERE column_name ***>value1*** AND column_name **= < *value2***;

or Equivalent to
WHERE column_name ***>= value1*** AND column_name ***=< value2***;

So you must test your database before using the between…and keyword
Examples

1.
```
SELECT *
FROM Employee
WHERE Age BETWEEN 23 AND 35;
```
Meaning: SELECT all records FROM the Employee table where the candidate is older than 23 and younger than 35

2.
```
SELECT *
FROM Employee
WHERE Department BETWEEN 'ARMY' AND 'FT';
```
Meaning: SELECT all records FROM the Employee table where Department is alphabetically greater than Doctor and less than Teacher.

3.
```
SELECT *
FROM Employee
WHERE Department NOT BETWEEN 'ARMY' AND 'FT';
```
Meaning: SELECT all records FROM the Employee table where Department is NOT alphabetically greater than Doctor and less than Teacher.

4.
```
SELECT *
FROM Employee
WHERE Age NOT BETWEEN 30 AND 40
ORDER BY FirstName;
```
Meaning: SELECT all records FROM the Employee table where the age is NOT BETWEEN 30 and 40 then present the results by First Name in alphabetical order.

You can also use **NOT BETWEEN** to exclude the values between your range.

IN Clause

The IN conditional operator is really a set membership test operator. That is, it is used to test whether or not a value (stated before the keyword IN) is "in" the list of values provided after the keyword **IN**.

SYNTAX

```
SELECT col1, SUM(col2)
FROM "list-of-tables"
WHERE col3 IN
       (list-of-values);
```

For example:

```
SELECT FirstName, LastName, Salary
FROM Employee
WHERE LastName IN ('Fox', 'Lulla', 'Phoenix', 'Bal');
```

This statement will select the FirstName, lastname, salary from the Employee table where the lastname is equal to either: Fox, Lulla, Phoenix, Bal. It will return the rows if it is ANY of these values.

The **IN** conditional operator can be rewritten by using compound conditions using the equals operator and combining it with OR - with exact same output results:

```
SELECT FirstName, LastName, Salary
FROM Employee
WHERE LastName = 'Fox' OR LastName = 'Lulla' OR LastName = 'Phoenix'
OR LastName = 'Bal';
```

As you can see, the IN operator is much shorter and easier to read when you are testing for more than two or three values.

You can also use **NOT IN** to exclude the rows in your list.

The Logical Comparison Statements → **AND …. OR**

AND - Combines two conditions and evaluates to **TRUE** when ***both*** of the conditions are **TRUE**.
OR - Combines two conditions and evaluates to **TRUE** when ***either*** condition is **TRUE**.

AND	Logical *and*
OR	Logical *or*

Examples

1.
```
SELECT *
FROM Employee
WHERE Age = 19 AND FirstName ='Neetu';
```
Meaning: SELECT all records FROM the Employee table WHERE age equals 19 AND FirstName = Neetu.

2.
```
SELECT *
FROM Employee
WHERE Age <=36 AND Department ='IT';
```
Meaning: SELECT all records FROM the Employee table WHERE age is less than or equal to 36 AND Department = IT.

3.
```
SELECT *
FROM Employee
WHERE Age !=36 AND Department >='IT';
```
Meaning: SELECT all records FROM the Employee table WHERE age NOT equal to 36 AND Department is alphabetically greater than or equal to IT.

4.
```
SELECT *
FROM Employee
WHERE Age <=36 AND Salary > 1000;
```
Meaning: SELECT all records FROM the Employee table WHERE age is less than or equal to 36 AND Salary is greater than 1000

5.
```
SELECT *
FROM Employee
```

```
WHERE Department <= 'IT'AND FirstName > 'Paul'
ORDER BY FirstName
```
Meaning: SELECT all records FROM the Employee table WHERE Department is alphabetically lesser than or equal to IT AND FirstName is alphabetically greater than Paul. Order by FirstName.

Using **Aliases**

Aliases are alternate names for a field or value. They are assigned by using the **AS** keyword.
SQL has two kind of **aliases** one for **columns** and the other for **tables**

SYNTAX for a column name ALIAS

```
SELECT column_name AS alias
FROM table_name ;
```

SYNTAX for table ALIAS

```
SELECT column_name
FROM table_name AS alias ;
```

We can use aliases in order to make column or table names more clear and meaningful.

Examples

Column Alias

```
SELECT FirstName As Name , Department As Career
FROM Employee ;
```

Table Alias

```
SELECT *
FROM Employee AS Person;
```

The **COUNT** function

The **COUNT** function is just one of the "**Aggregate Functions**" available to us in SQL.
We use the COUNT function to count the number of rows in a table, or to find the number of rows that match a specific criteria.

SYNTAX

```
SELECT COUNT ( )
FROM table_name ;
```

What do we put in the parenthesise? Well, we have 2 choices if we are using **ANSI SQL** (Standard Basic SQL) and a third if we are using SQL Server 2000 or Oracle.
So, in our first **ANSI** case we could use **(*)** in order to count how many rows are in a column.

SYNTAX

```
SELECT COUNT ( * )
FROM table_name;
```

Example 1

```
SELECT COUNT ( * )
FROM Employee ;
```

Example 2

In our 2nd example we put the **(column_name)** in the parenthesise in order to count how many rows are in a column contain a value.

```
SELECT COUNT (FirstName)
FROM Employee;
```

SUM, AVG, MAX , MIN functions.

Aggregate functions are used to compute against a "returned column of numeric data" from your SELECT statement. They basically summarize the results of a particular column of selected data. We are covering these here since they are required by the next topic, "GROUP BY". Although they are required for the "GROUP BY" clause, these functions can be used without the "GROUP BY" clause.

MIN	returns the smallest value in a given column
MAX	returns the largest value in a given column
SUM	returns the sum of the numeric values in a given column
AVG	returns the average value of a given column
COUNT	returns the total number of values in a given column
COUNT(*)	returns the number of rows in a table

The general syntax for using these Aggregate functions is

```
SELECT FUNCTION ( )
FROM table_name ;
```

SUM

We use the **sum** function to find the total sum of a column of **figures**, any **null** values are not included in the calculation

```
SELECT SUM ( column_name )
FROM table_name ;
```

Example

```
SELECT SUM (Salary)
FROM Employee ;
```

MAX

We use the **max** function to find the highest value of a column.

```
SELECT MAX(column_name )
FROM table_name ;
```

Example

```
SELECT MAX(Salary )
FROM Employee;
```

MIN

We use the **min** function to find the highest value of a column.

```
SELECT MIN ( column_name )
FROM table_name ;
```

Example

```
SELECT MIN(Age)
FROM Employee;
```

AVG

We use the **avg** function to find the **average** value of the figures in a column.

```
SELECT AVG( column_name )
FROM table_name ;
```

Example

```
SELECT AVG(Age)
FROM Employee;
```

Mathematical Operators

Standard ANSI SQL-92 supports the following first four basic arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

The modulo operator determines the integer remainder of the division. This operator is not ANSI SQL supported, however, most databases support it. The following are some more useful mathematical functions to be aware of since you might need them. These functions are not standard in the ANSI SQL-92 specs, therefore they may or may not be available on the specific RDBMS that you are using.

ABS(x)	returns the absolute value of x
SIGN(x)	returns the sign of input x as -1, 0, or 1 (negative, zero, or positive respectively)
MOD(x,y)	modulo - returns the integer remainder of x divided by y (same as x%y)
FLOOR(x)	returns the largest integer value that is less than or equal to x
CEILING(x) or CEIL(x)	returns the smallest integer value that is greater than or equal to x
POWER(x,y)	returns the value of x raised to the power of y

ROUND(x)	returns the value of x rounded to the nearest whole integer
ROUND(x,d)	returns the value of x rounded to the number of decimal places specified by the value d
SQRT(x)	returns the square-root value of x

For example:

```
SELECT round(Salary), FirstName
FROM Employee
```

This statement will select the salary rounded to the nearest whole value and the firstname from the employee table.

The **JOIN** Statement

How to deal with more than one table ?

Earlier, we learned how to deal with extracting data from a single table. If we want to deal with more than one table we have to use the **JOIN** statement.

Note: In order to make a join between multiple tables each table must contain a column that holds a unique value. This is known as the "Primary Key". Each table may contain only one primary key. A primary Key column can not contain a null value.

Example of primary key

Let's look at this table and try to find the primary key.

Person Table			
FirstName	**ID_No**	**Job**	**Age**
Vijay	01	Engineer	27
Sashi	02	Engineer	25
Prudhvi	03	Doctor	23
Praveen	44	Artist	30
Kiran	05	Teacher	35
Sashi	61	Politician	

In this table we can see that the columns Name and Job each of them each have repeated values. In the Job column engineer is repeated and in name column Sashi is repeated. In the Age column we have a null value for Sashi. It is also obvious that as our database grows we will have people on it of the same age, making the age column unsuitable as a primary key.

Explaining the foreign key

So let us examine the **Id_No**, in this column all the fields are unique (not repeated) and we don't have any Null values on it , so it is the primary key for this table. On the other hand, if we wish to use the column **ID_No** in another table which has it's own primary key we could designate it as a foreign key.

Getting data from multiple tables

In this example we have two tables, the **employee** (**emp**) table and the **department** (**dept**) table. We want to build an SQL statement in order to get some data from each of them at the same time. To do this, we are going to use the simplest kind of **join**.

Employee			
EmpID	**Name**	**Salary**	**DepartmentID**
01	Vijay	2700	20
02	Sashi	2500	20
03	Prudhvi	3000	30
04	Praveen	1950	30
05	Kiran	2100	10
06	Durga	1870	40
07	Sudhir	7000	

Department		
DepartmentID	**Name**	**Established**
10	Engineering	1993
20	Administration	1993
30	Research	1994
40	Head Quarters	1993
50	Human Resources	1994

SYNTAX

```
SELECT column_name.table1, column_name.table2
FROM table1, table2
WHERE table1.column_name= table2.column_name;
```

Example **1**

```
SELECT Employee.Name, Department.Name
FROM Employee, Department
WHERE Department.DepartmentID = Employee.DepartmentID
```

Example **2**

This example would return the same results as example 1 but we are using a table alias.
Remember, we sometimes use aliases to speed up or simplify our programming.

```
SELECT e.Name, d.Name
FROM Employee e ,Department d
WHERE e.DepartmentID = d.DepartmentID
```

There are three type of join → Inner, Left, Right joins

The INNER JOIN

SYNTAX

```
SELECT column_name.table1, column_name.table2
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name
```

Example

```
SELECT emp.Name, dept.DepartmentID , dept.Name
FROM Employee emp
INNER JOIN Department dept
ON emp.DepartmentID = dept.DepartmentID
```

The RIGHT JOIN

Supposing we wanted to see all the department names in the **dept** table even if they do **not** have any employees, to deal with this case we would use the **right join**.

SYNTAX

```
SELECT column_name.table1, column_name.table2
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name
```

Example

```
SELECT emp.Name, dept.DepartmentID , dept.Name
FROM Employee emp
RIGHT JOIN Department dept
ON emp.DepartmentID = dept.DepartmentID
```

The LEFT JOIN

So … if we want to see *all* of the employees and their departments and we *also* want to see the employee who does not have a department number (because I think he is the boss of this company) we use the left join.

SYNTAX

```
SELECT column_name.table1, column_name.table2
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Example

```
SELECT emp.Name, dept.DepartmentID , dept.Name
FROM Employee emp
LEFT JOIN Department dept
ON emp.DepartmentID = dept.DepartmentID
```

We can use the order by statement and the logical condition in the where statement but we can't make a logical condition in the inner join ,right join and left join.

Creating Tables

The **create table** statement is used to create a new table. Here is the format of a simple **create table** statement:

```
create table "tablename"
  ("column1" "data type",
  "column2" "data type",
  "column3" "data type");
```

Example**:**

```
create table employee
  (first varchar(15),
   last varchar(20),
   age number(3),
   address varchar(30),
   city varchar(20),
   state varchar(20));
```

To create a new table, enter the keywords **create table** followed by the table name, followed by an open parenthesis, followed by the first column name, followed by the data type for that column, followed by any optional constraints, and followed by a closing parenthesis. It is important to make sure you use an open parenthesis before the beginning table, and a closing parenthesis after the end of the last column definition. Make sure you seperate each column definition with a comma. All SQL statements should end with a ";".

The table and column names must start with a letter and can be followed by letters, numbers, or underscores - not to exceed a total of 30 characters in length. Do not use any SQL reserved keywords as names for tables or column names (such as "select", "create", "insert", etc).

Data types specify what the type of data can be for that particular column. If a column called "Last_Name", is to be used to hold names, then that particular column should have a "varchar" (variable-length character) data type.

Here are the most common Data types:

`char(size)`	Fixed-length character string. Size is specified in parenthesis. Max 255 bytes.
`varchar(size)`	Variable-length character string. Max size is specified in parenthesis.
`int`	Number value with a max number of column digits specified in parenthesis.
`datetime`	DateTime value

What are Constraints?

When tables are created, it is common for one or more columns to have **constraints** associated with them. A constraint is basically a rule associated with a column that the data entered into that column must follow. For example, a "unique" constraint specifies that no two records can have the same value in a particular column. They must all be unique. The other two most popular constraints are "not null" which specifies that a column can't be left blank, and "primary key". A "primary key" constraint defines a unique identification of each record (or row) in a table. All of these and more will be covered in the future Advanced release of this Tutorial. Constraints can be entered in this SQL interpreter, however, they are not supported in this Intro to SQL tutorial & interpreter. They will be covered and supported in the future release of the Advanced SQL tutorial - that is, if "response" is good.

Inserting into a Table

The **insert** statement is used to insert or add a row of data into the table.
To insert records into a table, enter the key words **insert into** followed by the table name, followed by an open parenthesis, followed by a list of column names separated by commas, followed by a closing parenthesis, followed by the keyword **values**, followed by the list of values enclosed in parenthesis. The values that you enter will be held in the rows and they will match up with the column names that you specify. **Strings should be enclosed in single quotes, and numbers should not**.

```
insert into "tablename"
 (first_column,...last_column)
  values (first_value,...last_value);
```

In the example below, the column name first will match up with the value 'Luke', and the column name state will match up with the value 'Georgia'.

Example:

```
insert into employee
  (first, last, age, address, city, state)
  values ('Luke', 'Duke', 45, '2130 Boars Nest',
          'Hazard Co', 'Georgia');
```

Note: All strings should be enclosed between **single** quotes: `'string'`

Updating Records

The **update** statement is used to update or change records that match a specified criteria. This is accomplished by carefully constructing a where clause.

```
update "tablename"
    set "columnname" = "newvalue"
           [,"nextcolumn" =
             "newvalue2"...]
    where "columnname"
      OPERATOR "value"
      [and|or "column"
       OPERATOR "value"]; [] = optional
```

Examples

1.
```
update phone_book
   set area_code = 623
   where prefix = 979;
```

2.
```
update phone_book
   set last_name = 'Smith', prefix=555, suffix=9292
   where last_name = 'Jones';
```

3.
```
update employee
   set age = age+1
   where first_name='Mary' and last_name='Williams';
```

Deleting Records

The **delete** statement is used to delete records or rows from the table.

```
delete from "tablename"
where "columnname"
  OPERATOR "value"
[and|or "column"
  OPERATOR "value"];
[ ] = optional
```

Examples:

```
delete from employee;
```

Note: if you leave off the where clause, **all records will be deleted!**

```
delete from employee
  where lastname = 'May';

delete from employee
  where firstname = 'Mike' or firstname = 'Eric';
```

To delete an entire record/row from a table, enter "delete from" followed by the table name, followed by the where clause which contains the conditions to delete. If you leave off the where clause, all records will be deleted.

Drop a Table

The **drop table** command is used to delete a table and all rows in the table.
To delete an entire table including all of its rows, issue the **drop table** command followed by the tablename. **drop table** is different from deleting all of the records in the table. Deleting all of the records in the table leaves the table including column and constraint information. Dropping the table removes the table definition as well as all of its rows.

```
drop table "tablename"
```

Example:

```
drop table myemployees_ts0211;
```