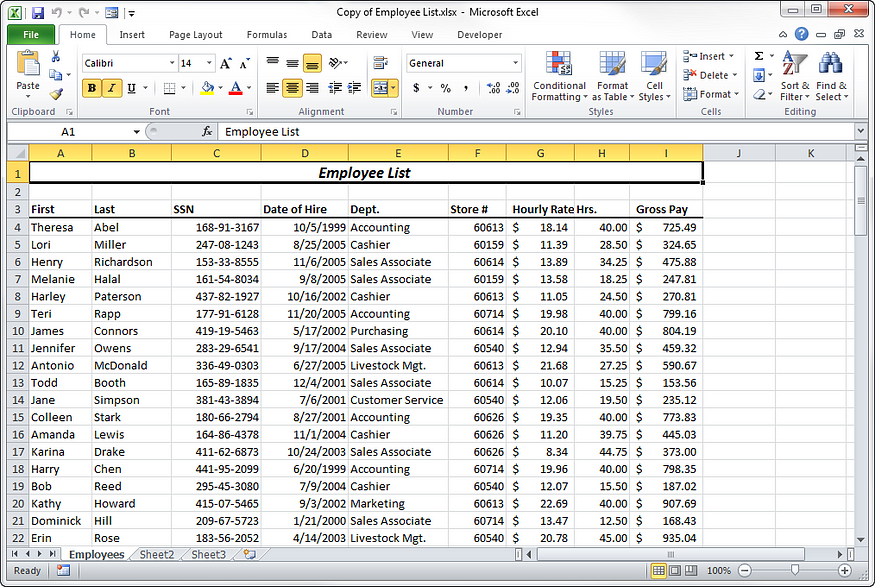
**Why Do We Need A Database?**

When you have some data, and you want to store this data somewhere. This data could be anything. It could be about customers, products, employees, orders, …etc. This data could be in text format, numeric, dates, document files, images, audio, or video.  
  
Maybe if you have data about the customers in your company, the first thing that comes to mind is to open a spreadsheet. Then you start writing whatever data you want to store.  
  
It could be the customer name, id, position, and so on. You may add as many as customers, delete any of them later, or even modify them. And that’s it!.

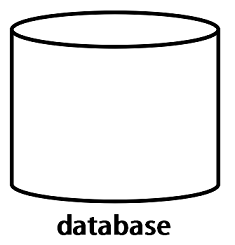


Probably this is what comes into your mind when you hear the term “database”.

Now, we have a kind of data, and you stored them in spreadsheets in a way that satisfies your need. That might be OK, because just having data is not a good enough reason to need a database, and it’s not the problem. The problem is what comes next, and there’s a lot of potential problems.

What if you have a bunch of data, maybe 10,000 customers, “*Are you going to scroll down in the spreadsheet to get the 9999 customers?!*”, What if the security was a concern, “*Do you care about if someone else got access to your data?*”, What if you accidentally put redundant information, “*Is it fine to have duplicate information along with the spreadsheet?”.*  
  
This takes us to the next question, “*When do we actually need a database?*”. Consider the following potential problems:  
  
**Size**You may have thousands or millions of rows of customers, or any piece of information. **Accuracy***“Do you care if someone entered incorrect data?”.* If yes, nothing could actually prevent me from typing incorrect data into a spreadsheet. **Security**  
If the data is sensitive, and you need to restrict access to the data; It doesn’t need to be shared with everyone. In addition, *“Do you need to know who made every change at every point?”*. **Redundancy**If the redundant data (having multiple copies of the same data) will lead to conflict, you would need to have only non-repeated unique data. **Importance***“What if you had a disconnect or a crash, and you lost your data?”.*You’ve probably felt that pain before. And it’s unacceptable to lose important data like orders of a customer, allergies of a patient, flight bookings, …etc. **Overwriting**How about having more than one person overwriting the same data at the same time. How about 10 at the same time or 100 people at the same time?. You’ll end up with everybody overwriting everybody else’s changes.If you are saying “Yes” to one of these problems, or all of them and more besides, to keep the data reliable, secured, and maintainable. So, you need to have a database. That’s what we are going to discuss here.

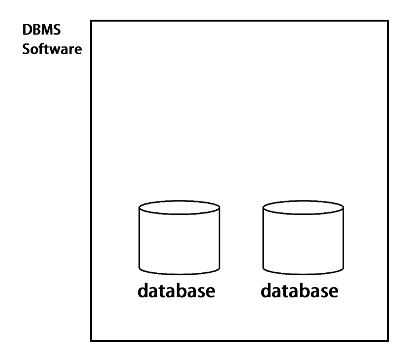
**What’s A Database?**



It’s a structured system to put your data in that imposes rules upon that data, and the rules are yours because the importance of these problems changes based on your needs. Maybe your problem is the *size*, while someone else has a smaller amount of data where the *sensitivity*is a high concern.  
It’s the things you can’t see that is going on in the background; the security, the enforced integrity of the data, the ability to get to it fast and get to it reliably, the robustness; serving lots of people at the same time and even correctly survive crashes and hardware issues without corrupting the data.  
And that’s what we need to do here; understand how to describe our structure and define those rules, so all these invisible things will actually happen.

# Database Management System (DBMS)

We often mistakenly say our database is Oracle, MySQL, SQL Server, MongoDB. But, they aren’t databases, they are database management systems (DBMS).  
  
The DBMS is the software that would be installed on your personal computer or on a server, then you would use it to manage one or more databases.  
  
The database has your actual data and the rules about that data, while the DBMS is the program that surrounds and manages your actual data, and it enforces the rules you specified on your data. The rules for example could be the type of the data, like integer or string, or the relationship between them.



In practice, it’s very common to have multiple databases. The database that deals with your order and customer information might be completely independent of your database that deals with human resource information. And in many organizations, you don’t just have multiple databases but multiple DBMS. Sometimes it’s because one DBMS is better at something than the other.  
  
There are different DBMS, and they are categorized under:  
  
Relational Database Management Systems  
Hierarchical Database Systems  
Network Database Systems  
Object-Oriented Database Systems  
NoSQL Database Systems  
  
We are going to focus on relational database management systems (RDBMS). And here’s *Why? …*  
  
They are the most commonly used ones.  
The principles we are going to discuss here are usable across all of them.  
If you know you are going to jump into NoSQL databases, most of the introductions assume you already understand relational database concepts and will use these concepts to explain what’s offered by NoSQL databases.  
  
*RDBMS are like Oracle, MySQL, SQL Server, SQLite, DB2, …etc.*

# Database & Tables

## Database

A database is a collection, or a set of tables. Each table has a formalized repeating list of data about one specific piece of information. For example a table for customers, students, orders, products, and so on. Visually, it’s often shown like a spreadsheet.

## Tables

The table is the most basic building of a database. It’s the place where you will put your data, define their data type, and also their relationship with the other tables. It consists of rows and columns.

# Rows & Columns

Within each table, every single **row**represents one single student, customer, order, or employee. But each of these rows is not free form. You must apply structure to this data. So, you must say what every row is made of, and you do this by defining the **columns**in that table. And each column describes one piece of data. It gives it a **name**like name, id, email, date of birth, and a **type**, perhaps, a text, or a date, or a number.  
  
Now, every row must follow that same structure, following that same format. It’s not allowed to deviate from the way that the columns are set up. And by defining these columns, we’re imposing rules on the data, and the DBMS won’t let us break them.  
 **In a nutshell,** columns define what’s the data that should be in the table, while the rows hold the actual values that you are going to retrieve, insert, update, and delete.  
  
*You may hear the term “tuple” instead of rows, and also you may hear the term “attribute” instead of column.*  
  
Before heading into the next topic, there’s something to be mentioned about columns, and that’s there are 3 types of columns:

## 1. Simple

It’s just a single value.

## 2. Composite

A value that’s composed of some other values. For example, you may have a name that’s composed of first name, middle name, and last name. *Any composite attribute will be decomposed into separate simple attributes.*

## 3. Multi-valued

Multiple values for a single column. For example, the color of the car may be black and red.

# Primary Key

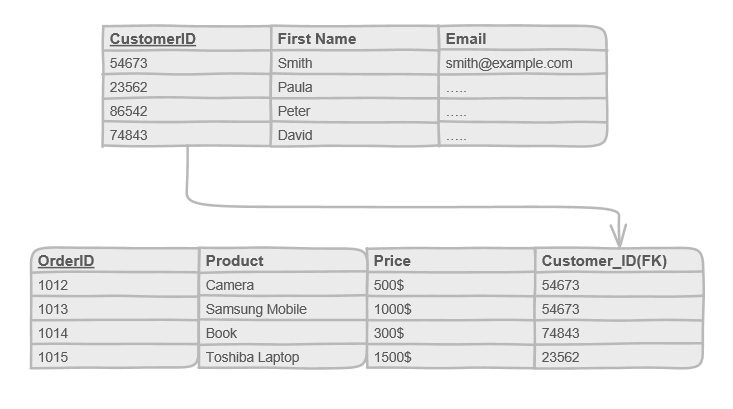
Now, if you have a long list of rows, it’s essential to have something that uniquely identifies each row, and that’s called the “primary key”. A primary key is a column of unique values for each row.  
  
You may have more than one student with the same name, but you can’t have more than one student with the same primary key. And if you tried to insert a duplicate value, this will be disallowed by the DBMS.  
  
*Usually you will see a primary key column called “id” of integer values.*  
  
The primary key is either naturally exists or generated by the DBMS. It means, for example, by default every customer has a unique SSN, assigned by the company. So, the SSN uniquely identifies every customer.  
But, you may have a table that holds information about some products, and they don’t have ids by nature. So, you will ask the DBMS to generate a new unique column, like product id and you may want to mark it as “auto-increment”.

*Auto-increment columns allows a unique number to be generated when a new record is inserted into a table by increment the value by 1 for each new record.*Primary keys are very important, not only to uniquely identify the rows, but also we are going to use them to connect between the tables and form relationships.

**One-to-Many Relationship**

Most of your tables will be naturally be connected, so we need to have a relationship between them. You’re not trying to invent relationships that don’t exist, you’re trying to describe what’s already there.

An example of a one-to-many relationship could be, a customer can place more than one order, but, an order is only placed by one and only one customer. You can’t have an order that’s placed by more than one customer.  
  
The customer’s information exists in a table, and the order information also exists in another table, but they have a relationship. So, “*How can we define a one-to-many relationship?*”.  
Do you remember when we said primary keys are used to connect between tables? So, here we will add a new column to the orders table (many side) called “foreign key”. This column has the primary key values of customers.



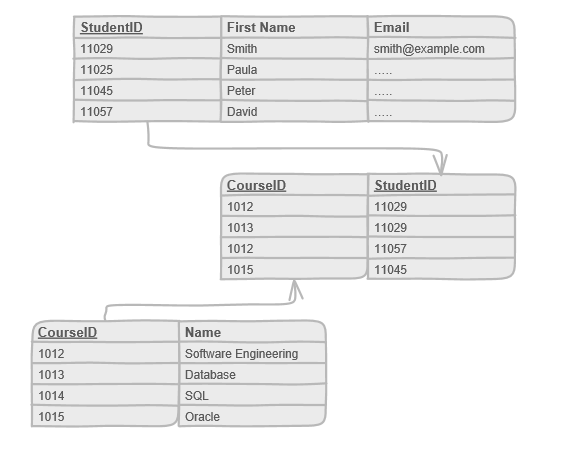
One-to-Many Relationship

The values of the foreign key column could be redundant because a customer can place more than one order. And a customer can have one, or more or even nothing number of orders.

The benefit of defining this relationship is, it allows us to ask some questions like: “*What are the orders placed by a customer whose first name is “Smith”?*”, or go the other way, “*Who is the customer who placed the order of id 1012?*”.

# Many-to-Many Relationship

You will be using a one-to-many relationship a lot between your tables. But, there is another way to relate tables together. What if you have a student, and each student can be enrolled in one or more courses, and at the same time, each course can have one or more students enrolled in it.  
  
So, it’s no longer a one-to-many, it’s many from both sides. And, as usual, the primary keys are the way to connect the tables. But, this time we will create a new table. It’s usually called a “junction” or “linking” table.  
  
This table exists only to connect the two tables, it has two foreign keys, one points to the primary key of the students table, and the second one points to the primary key of the courses table.  
And the two foreign keys together will form the primary key of the new table.



Now, using this relationship, we can know the courses of a particular student, or go the other way and get the students who are enrolled in a specific course.

# Structured Query Language (SQL)

It’s the language that’s used to create, read, update or delete data (falls under the acronym CRUD), and also to define the databases themselves.  
  
It’s not a programming language, it’s considered as a declarative query language. You just need to say what you want, and you let the DBMS handle how that’s actually done for you.  
  
On the other hand, in the programming language, you would have to do write the steps to do this; maybe some loops, and check to see if this is the required data or not.

# Database — Design Process

# Database Design Process

The processes here aren’t the same as the agile model or iterative approach. They are defined steps to end up having a fully defined database, with its constraints, and structure.  
There is no place for many changes because they are going to cost you a lot. So, you need to be specific and take things step by step. And, here’s the steps:

## 1. Requirements Gathering

Understanding what you want to do, and what you have is essential before you can dive into designing a database. We’ll look at the steps in this article.

## 2. Conceptual Design

We specify the entities, columns, and their relationship. We may use an entity relationship (ER) diagram to visualize the database.  
  
**The output is:**A conceptual schema (described using a conceptual data model like ER model).

## 3. Logical Design

It’s concerned about data model mapping; mapping a conceptual schema (like ER model) into logical schema to provide a much detailed description.  
  
**The output is:** A logical schema (described using a logical data model specific to the DBMS like relational model).

## 4. Physical Design

It describes the details of how data is stored. You start by defining (already modeled) tables, how the data is stored, define relationships, … in DBMS. This requires dealing with the DBMS, and could involve SQL.

**The output is:** An internal (physical) schema (described using a physical data model).  
  
*A database****schema****is a description of a database structure, data types, and the constraints on the database.*

# Requirements Gathering

The first step in whether you are building a mobile application, desktop, or any kind of software is to gather the requirements about what’s actually needed. And, here are the steps:

## 1. Why? and is it feasible?

So, first, you study “Why we need a database?”, and if it’s feasible or not. You need to check if it could be implemented under the current budget, under the current technical skills of your team, within the defined schedule, And if it does contribute to the whole organization's objectives or not.

## 2. Collecting the requirements

If the answer was Yes!, then you start collecting the requirements. You can do this by having interviews with all the stakeholders; anyone who will use the system. And try to get any kind of existing data, any kind of spreadsheets, this will help you to in building the database.And if there is an existing database, then ask, what’s wrong with it? what’s right with it? These questions will help to avoid any of the current problems.

## 3. Group related requirements

Now, you may need to group related requirements together, resolve any conflicts, or any ambiguity by negotiation with the customers.

**4. Specify & Verify the requirements**

Finally, you specify the requirements, or in other words, you write the requirements in a document, and validate the requirements with the customer to make sure that everything is on the right track.

# Database — Normalization

# The process of organizing your database through a set of defined steps.



This is a process where you take your database design, and you apply a set of formal criteria of rules called “Normal Forms”. And we step through them first normal form, second normal form, and third normal form.  
You usually will end up creating a few new tables as part of the process. But, the end result of normalization is to make your database easier to edit, easier to maintain, and preform operations, remove any duplicates, & more reliable to work with.  
  
**Denormalization**

Sometimes you will hit the situation where you will break the normalization rules for performance improvements, because normalization is often involves splitting data into multiple tables.  
  
If you are following along, you remember at the first normal form, when we explained the example of emails. “What happens if we added two email columns to the customer table?.”  
  
Technically, this can be described as breaking first normal form. It’s a repeating group. But in practice, you may find it more convenient to just allow an email1 and email2 columns, because you’re sure there won’t be a flexible number of email addresses.  
  
This will improve your performance, and save you from creating more tables.

# Database — Structured Query Language (SQL)

**SQL Syntax**

A SQL statement begins with a keyword, like *SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE* and ends with a semicolon “;”.

**SELECT** *column1, column2, ..., columnN* FROM *table\_name***;**

**Case Sensitivity**SQL is not case sensitive, but, most developers write the SQL keywords all uppercase, because it makes these statements easier to read. **White Space**The line breaks also don’t matter. SQL is not sensitive to white space. You can split a SQL statement into multiple lines for readability. **Semicolon**Officially, a complete SQL statement should be ended with a semicolon, though, many DBMS just don’t care.

**Comments**You can write comments in two ways; either using the c-style comment, or using two hyphen characters (with no space between them) followed by a space and the comment text.

**/\*   
 this is a  
 multi-line comment  
\*/**  
SELECT \* FROM Employee; **-- this is a single line comment**

**Operators**

An operator is a reserved word used to perform an operation like combining between expressions, comparisons, or arithmetic operations.

* **Arithmetic:** +, -, \*, /, and %.
* **Comparison:**=, !=, >, <, ≥, ≤, <>, …etc.
* **Logical:** AND, OR, NOT, …etc.

The SQL language can be divided into data definition language (DDL), data manipulation language (DML), and data control language. We’ll get into everyone of them. So, let’s get started.

**Data Manipulation Language (DML)**

The data manipulation language is used to *create*, *read*, *update*, and *delete*data from a table. They are grouped together under the acronym CRUD.  
  
The DML operations are the majority of SQL that you’ll write. So, it’s worthwhile to take a deeper look at them.  
  
**SELECT**  
  
The SELECT statement is used to retrieve data. The results may come from one or more table, or even expressions.

**SELECT** ‘Hello World’;  
**SELECT** 1 + 2;  
  
Now, let’s select some rows from a table in the database.  
  
The format is to specify the columns you want separated by commas, and then the FROM clause followed by the table name.

**SELECT** firstName, lastName **FROM** Employee;  
  
*The columns are displayed in the ordered we specified in the SELECT statement.*  
  
In order to return all the columns in a table, you can use the *asterisk*.

**SELECT \*** **FROM** Employee;

**WHERE Clause**You can get data that only meet a specific condition, instead of getting the entire table. The WHERE clause is used to filter the result data.For example, we can select only employees who have *salary*> 10,000.

SELECT \* FROM Employee **WHERE salary > 10000;**  
  
Or, those who have *last name* equals to ‘Fuller’.

SELECT \* FROM Employee **WHERE lastName = 'Fuller'**;  
  
*In SQL, a string must be in a single quotes, not the double quotes.*  
  
You can combine conditions by using the logical operators *AND & OR*.

SELECT \* FROM Employee   
**WHERE salary > 10000 AND lastName = 'Fuller'**;  
  
*Rows are returned only if the boolean expression in the WHERE clause returned TRUE for each row.*

**Comparing NULL**

If some *employees’ last name*are given NULL instead of real values, those employees won’t be included in the result unless you use IS (vice-versa is IS NOT) operator in the WHERE clause to compare to NULL (missing data).

SELECT \* FROM Employee   
WHERE lastName = 'Fuller' OR **lastName IS NULL**;  
  
*We can’t use the equal sign with NULL when comparing column’s value. Instead, use the operators; IS NULL, and, IS NOT NULL. Keep in mind that NULL****is not a value****, it’s a state of no value, not zero, not empty string.*

**Aliases**SQL aliases are used to give a temporary name to a table, or a column in your result set using the AS operator.

SELECT lastName **AS lname** FROM Employee; -- for column  
SELECT \* FROM Employee **AS emp**; -- for table

The AS operator can be useful when:  
  
There are more than one table involved in a query.  
Functions are used in the query.  
Column names are big or not very readable.  
Two or more columns are combined together.

**ORDER BY Clause**

SQL does not guarantee that data is returned in any particular order unless you specify an order. This can be done using ORDER BY clause to return the data ordered by one or more columns.

SELECT \* FROM Employee **ORDER BY** **salary**;

The default order is in ascending order. If you want them in descending, we can just put DESC (or ASC for ascending — default) after ORDER BY clause.

SELECT \* FROM Employee **ORDER BY** **salary** **DESC**;  
  
We can also order by more than one column. It will sort them by the first column, *department*in this case, and then for each *department*, it will sort them by *last name*.

SELECT \* FROM Employee **ORDER BY** **deptName, lastName**;  
  
Sure, you can use the descending order for *department*instead of the default.

SELECT \* FROM Employee **ORDER BY** **deptName DESC, lastName**;

## DISTINCT

We can select only distinct values of a column using SELECT DISTINCT statement. The following query will return all unique values for last name in the employee table.

**SELECT DISTINCT** **lastName** FROM Employee;  
  
*The DBMS will actually sort the data, and move out the duplicates.*Using more than one column will result in having duplicates, but, the whole row of columns’ values will be distinct.

**SELECT DISTINCT** **lastName, firstName** FROM Employee;

## LIKE Operator

You can filter the columns based on a specified text. For example, we can select the employees who have “California” some where in the address.  
  
The LIKE operator is used in the WHERE clause to match a specific pattern in a column value.

SELECT \* FROM Employee WHERE address **LIKE '%California%'**;

*The percent sign “%” is a wildcard that means zero or more characters. A wildcard character can be used to substitute for other character(s) in a string.*So, we can also match only those employees where their address starts with ‘California’ by removing the percent sign at the beginning of the string.

SELECT \* FROM Employee WHERE address **LIKE 'California%'**;

Other wildcards like the underscore “\_” can substitute a single character.

The next query will return all the employees who have the following format in their address: Any letter in the first position, letter ‘a’ in the second position and anything after it.

SELECT \* FROM Employee WHERE address **LIKE '\_a%'**;  
  
***Regular expressions****are also used to match a particular pattern. They are much more powerful and complex than LIKE patterns. You need to look closely on how your database system implements regular expressions.*

## IN Operator

The IN operator used to match a set of possible values in the WHERE clause.

SELECT \* FROM Employee   
WHERE lastName **IN ('Maria', 'Thomas', 'Fuller');**

Likewise, use NOT IN to select the values that don’t match a set of values.

SELECT \* FROM Employee   
WHERE lastName **NOT IN ('Maria', 'Thomas', 'Fuller');**

## BETWEEN Operator

The BETWEEN operator is used to select values within a range.

SELECT \* FROM Employee  
WHERE salary **BETWEEN 2000 AND 3000**;

Likewise, NOT BETWEEN selects the values that are outside the range.

SELECT \* FROM Employee  
WHERE salary **NOT BETWEEN 2000 AND 3000**;

*The behavior of BETWEEN operator is different from one database management system to another. So, check how it’s being treated by your database system.*

## Functions

SQL built-in functions are used to perform specific operations on data. A list of all the functions can be found [here](https://www.tutorialspoint.com/sql/sql-useful-functions.htm).

## — Aggregate

Those functions aggregates the rows, resulting in one value (unless we use GROUP BY clause which we will cover here).

**-- COUNT  
--** Counts the number of rows.  
SELECT **COUNT(\*)** FROM Employee;**-- SUM  
--** The sum of the salary of all the employee.  
SELECT **SUM(salary)** FROM Employee;**-- MIN & MAX**  
-- Selects the min or max salary.  
SELECT **MAX(salary)** FROM Employee;

## — Numeric

SQL numeric functions are used for numeric manipulation and mathematical calculations.

**-- ABS  
--** Returns the absolute value of a number.  
SELECT **ABS(salary)** FROM Employee;**-- POWER  
--** Finds the value of 1st arg raised to the power of 2nd arg.  
SELECT **POWER(salary, 2)** FROM Employee;**-- ROUND**  
-- Rounds a number to the nearest integer.  
-- If 2nd arg is passed, then round to 2nd arg decimal places.  
SELECT **ROUND(salary, 2)** FROM Employee;

## — String

SQL string functions are used to process and manipulate a string.

**-- LENGTH  
--** The length of a column of a string value.  
SELECT **LENGTH(address)** FROM Employee;**-- SUBSTR  
--** Selects a part of the string (1-indexed)  
**SELECT SUBSTR(address, start, len)** FROM Employee**;-- TRIM, LTRIM, RTRIM**  
-- Removes the leading spaces either from both sides or one side.  
SELECT **TRIM(address)** FROM Employee;**-- UPPER & LOWER**-- Converts characters to upper or lower case.  
SELECT **UPPER(address)** FROM Employee;

*The standard SQL doesn’t include LENGTH or SUBSTR, but, most DBMS do. The names might be different from one database system to another, but, this is generally how they work.*

## Stored Functions

A stored function (also called a user-defined function) is a function stored in the database, and can be called same as any built-in functions.  
  
The implementation can vary from one database system to another, but, to get a quick idea on they’re implemented, here is an example in MySQL.

-- Create a function  
**CREATE FUNCTION** sayHello (name CHAR(30))  
**RETURNS** CHAR(50) **DETERMINISTIC**  
 **RETURN** CONCAT('Hello, ', name, '!') **;**-- Use the function  
SELECT **sayHello(lastName)** FORM Employee;  
  
*A function is considered “deterministic” if it always produces the same result for the same input parameters, and “not deterministic” otherwise.*

## GROUP BY Clause This clause is used (usually in conjunction with the aggregate functions) to group the resulting rows by one or more columns. We can get the number of employees grouped by their last name.

SELECT lastName, COUNT(\*)   
FROM Employee   
**GROUP BY lastName;**

## HAVING Clause

The HAVING clause was added to SQL because the WHERE clause couldn’t be used with aggregate functions.

The following query will find if there are more than 30 employee with the last name ‘Fuller’ or ‘Walter’.

SELECT lastName, COUNT(\*)   
FROM Employee  
WHERE lastName = 'Fuller' OR lastName = 'Walter'  
GROUP BY lastName  
**HAVING COUNT(\*) > 30;**

## JOIN

We have been selecting on a specific tables, but, what if we have tables that have some kind of relationship, whether 1-M, or M-M, and we want to connected those tables, and retrieve matching values.  
  
We don’t want to be limited to selecting from one table, but, to be able to select from two different tables or even three or more and the query we’re going to use is to JOIN our tables together.

## — INNER JOIN

For example, in 1-M relationship, let’s say we have a customer table, that can place one or many or no orders, where every order is placed by one customer. We can get the information of the customers and the orders placed by each customer.  
  
We join the two tables by matching the foreign key in the orders table, and the primary key of the customers.

SELECT Customer.ID, Customer.name, Order.number  
FROM Customer **INNER JOIN** Order  
**ON** Customer.ID = Order.CustomerID

*You need to use the dot ‘.’ sign to refer to a specific table as we did here. This is useful to avoid any conflict.  
  
Even thought the default kind of JOIN is an INNER JOIN (you can use the word JOIN instead of INNER JOIN), but, you should use INNER JOIN instead.*

## — LEFT & RIGHT OUTER JOIN

What we’ve done is called an INNER JOIN, meaning, only bring back the rows where there is a match in both tables.  
Sometimes you might want to start involving these other rows that don’t exactly match. We are interested in where they match but we still want to get the results where they don’t.  
So, we would use the word LEFT OUTER JOIN or RIGHT OUTER JOIN. You are typically saying one of these tables takes precedence over the other; Using the LEFT will also show customers that didn’t have any orders, while using RIGHT will also show all the orders that didn’t placed by any customer.

SELECT Customer.ID, Customer.name, Order.number  
FROM Customer **LEFT** **OUTER JOIN** Order  
**ON** Customer.ID = Order.CustomerID

*We will get NULL for the values that doesn’t exist, as a result of retrieve rows that doesn’t match.*

## — FULL OUTER JOIN

There is also a FULL OUTER JOIN which does both. It says to include all rows from both tables in the result, matching them whenever possible, but, returning rows with NULL when there’s no match between them.

**UNION Operator**

The UNION operator combines the result of two or more SELECT statements. There are some considerations to keep in mind when using UNION:

* Each SELECT statement in UNION must have same number of columns.
* The columns must also have similar data types.
* The columns in each SELECT statement must be in the same order.

The following SQL statement selects all the different birth dates from the *employee*and *depdendet*tables.

SELECT bdate **As 'birthDate'** FROM Employee  
**UNION**  
SELECT birth\_date **As 'birthDate'** FROM Dependent;  
  
*The column names in the result are usually equal to the column names in the first SELECT statement unless you give each similar two columns a specific alias using the AS operator.*The UNION operator selects only distinct values by default. To allow duplicate values, use the ALL keyword with UNION.

SELECT bdate **As 'birthDate'** FROM Employee  
**UNION ALL**  
SELECT birth\_date **As 'birthDate'** FROM Dependent;

# INSERT

The INSERT INTO statement is used to add one or more row to a table.

**INSERT INTO** Employee (lastName, firstName)   
**VALUES** ('Jason', 'Albert'), ('Walter', 'Savitch'), ('Christopher', 'John');  
  
*If a column is not assigned values, the default value for the column is used.*We can also insert all column values at once.

**INSERT INTO** Employee **VALUES** ('Jason', 'Albert', 14054, '123 Statham Street, California');

*The****number****of columns and values must be the same, and the inserted values must match the columns’****order****and****type****.*You can use the results of a SELECT statement in place of the VALUES clause. The SELECT statement will execute the query and return the result.

**INSERT INTO** backup (lastName, firstName) -- specific columns  
**SELECT lastName, firstName FROM Employee**;**INSERT INTO** backup **SELECT \* FROM Employee**; -- all columns  
  
Similarly, you can use the SELECT INTO statement to achieve the same task.

**SELECT** \* **INTO newtable [IN externaldb]** FROM table;

# UPDATE The UPDATE statement is used to change data in existing rows in a table. The new values are assigned using the SET keyword.

**UPDATE** Employee   
**SET** salary = 5000, address = '77 Winchester Lane, California'  
WHERE SSN = 5123;  
  
*Usually, you’ll use the WHERE clause to update(or delete) specific row(s). A good practice is to select first the rows you want to modify, make sure they are the expected rows and then perform update(or delete).*We can also assign a column to NULL instead of a real value.

**UPDATE** Employee **SET address = NULL** WHERE SSN = 5123;  
  
*If you want to assign a column value to NULL, use the equal sign.*

# DELETE The DELETE statement is used to delete one or more rows from a table. The format is DELETE FROM, then a table name, and usually a condition.

**DELETE FROM** Employee WHERE firstName = 'Henry';  
  
To delete all the rows from your table, you don’t need to use WHERE clause.

**DELETE FROM** Employee;

*You need to take care with your delete statements. DBMS won’t confirm back if you really want to delete those rows or not!.   
  
So, make sure you’re selecting the rows you expect to be deleted (or updated).*

# UPDATE

The UPDATE statement is used to change data in existing rows in a table. The new values are assigned using the SET keyword.

**UPDATE** Employee   
**SET** salary = 5000, address = '77 Winchester Lane, California'  
WHERE SSN = 5123;

*Usually, you’ll use the WHERE clause to update(or delete) specific row(s). A good practice is to select first the rows you want to modify, make sure they are the expected rows and then perform update(or delete).*

We can also assign a column to NULL instead of a real value.

**UPDATE** Employee **SET address = NULL** WHERE SSN = 5123;

*If you want to assign a column value to NULL, use the equal sign.*

# DELETE

The DELETE statement is used to delete one or more rows from a table. The format is DELETE FROM, then a table name, and usually a condition.

**DELETE FROM** Employee WHERE firstName = 'Henry';  
  
To delete all the rows from your table, you don’t need to use WHERE clause.

**DELETE FROM** Employee;  
  
*You need to take care with your delete statements. DBMS won’t confirm back if you really want to delete those rows or not!. So, make sure you’re selecting the rows you expect to be deleted (or updated).*

## Table

The CREATE TABLE statement is used to create a table in a database.  
Now, If you want to create a new table, you then specify the name of the columns, their datatype, and any other constrains inside the parenthesis.

CREATE TABLE Employee (  
SSN INTEGER,  
firstName VARCHAR(45),  
lastName VARCHAR(45),  
salary INTEGER,  
address VARCHAR(255),  
bdate DATE  
)

*The definition for each column is separated by a comma, except the last one.*

**Data Types**

A data type defines a set of possible values a column can contain.  
The data types that are available in one database management system are much different than the data types available on another database system.  
  
MySQL for example, supports three fundamental types of data:

1. **Numeric types** are used to represent numerical values.  
     
   [**Integer**](https://dev.mysql.com/doc/refman/en/integer-types.html)like *INTEGER*.  
   [**Fixed Point**](https://dev.mysql.com/doc/refman/en/fixed-point-types.html)like *DECIMAL*and *NUMERIC*.  
   [**Floating Point**](https://dev.mysql.com/doc/refman/en/floating-point-types.html) like *FLOAT*and *DOUBLE*.

**2. String types** are used for representing both text and non-text strings.

[**Fixed Length**](https://dev.mysql.com/doc/refman/en/char.html) like *CHAR*.  
[**Variable Length**](https://dev.mysql.com/doc/refman/en/char.html) like *VARCHAR*.  
[**Binary Strings**](https://dev.mysql.com/doc/refman/5.7/en/binary-varbinary.html) like *BINARY*and *VARBINARY.*  
[**Large Object Storage (BLOB and TEXT)**](https://dev.mysql.com/doc/refman/en/blob.html) like *BLOB*and *TEXT*.

**3. The date and time types**are used to represent temporal values.

* MySQL has five date and time types; [DATE, TIME, DATETIME, TIMESTAMP, and YEAR](https://dev.mysql.com/doc/refman/en/date-and-time-types.html).

**Table Constrains**

As you’re defining your table you may want to define specific rules and behaviors for some of your columns. This is done by using constraints. These constrains can’t be violated, and if you tried, your DBMS will complain. The constraints can be specified when creating or updating a table. In SQL, we have the following 6 constraints:

**— — 1. NOT NULL**

Indicates that a column can’t have NULL. It can’t be empty. It must have a value.

**— — 2. UNIQUE**

Ensures the values of a column (or combination of columns) are unique.

*A column with more than NULL doesn’t violates the UNIQUE constrain, because NULL is not a value.*

**— — 3. PRIMARY KEY**

A combination of a NOT NULL and UNIQUE; It ensures that a column (or combination of columns) have a unique values, and aren’t NULL.

*You can have many unique columns per table, but only one primary key is allowed.*

**— — 4. FOREIGN KEY**

Ensures a foreign key can’t have a value that’s not in the primary key it points to, but, it can have NULL (if allowed) just fine.

*What if we tried to insert a value in the foreign key that’s not in the primary key?*  
— It will be refused, and It’s not allowed, and you’ll get a constrain violation.

*What if we tried to delete or update a primary key that’s being referenced by a foreign key?*  
— It won’t be allowed by default, and you’ll get a constrain violation.

But, there are some other options that might be available by your database system like:

* **Cascade:** Cascade down and delete or update all referencing rows.
* **Set NULL:**Set the foreign key column to NULL.
* **No Action:** Refuse the delete or update operation (the default in most database management systems).

**— — 5. CHECK**

Ensures that the value in a column meets a specific condition.

**— — 6. DEFAULT**

Specifies a default value for a column.

*What if we when we insert a row, we didn’t enter the value of a column?*

The implementation of these constrains can vary from one database system to another. As an example, here’s the syntax in MySQL.

CREATE TABLE **Departement** (  
 number INTEGER **NOT NULL AUTO\_INCREMENT**,   
 name VARCHAR(45) **NOT NULL**,  
 **UNIQUE KEY (name),**  
 **PRIMARY KEY (number)**);CREATE TABLE Employee (  
 SSN INTEGER **NOT NULL**,  
 firstName VARCHAR(45) **NOT NULL**,  
 lastName VARCHAR(45) **NOT NULL**,  
 salary INTEGER **DEFAULT 0**,  
 address VARCHAR(255),  
 bdate DATE **NOT NULL**,  
 deptNumber INTEGER **DEFAULT NULL**,  
 **CHECK (SSN > 0)**, **PRIMARY KEY (SSN),  
 FOREIGN KEY (deptNumber) REFERENCES Departement (number)**);

# ALTER

## Database

The ALTER DATABASE statement is used to modify an existing database. Here’s an example in MySQL to update the character set and collation.

**ALTER DATABASE** Employee **CHARACTER SET** utf8 **COLLATE** utf8\_general\_ci;

## Table

Sometimes, you may need to change the definition of a table after it’s already been defined, and even filled up with data, like add, modify, or delete existing columns. You can do this with the ALTER TABLE statement.

-- To add a new column.  
  
**ALTER TABLE** Employee **ADD** email VARCHAR(255) ;-- To add a new column after a specific existing column.  
 **ALTER TABLE** Employee **ADD** email VARCHAR(255) **AFTER** SSN;-- To add a new column at the beginning.  
 **ALTER TABLE** Employee **ADD** email VARCHAR(255) **FIRST**;/\* New columns are filled with NULL by default.   
 You can specify a different default value. \*/  
 **ALTER TABLE** Employee **ADD** email VARCHAR(255) **DEFAULT ''**;/\* To modify an existing column   
 Database systems vary in the implementations of modifying a column in a table. Here is the implementation of MySQL. \*/  
 **ALTER TABLE** Employee **MODIFY** **COLUMN** email **VARCHAR(45) NOT NULL**;-- To remove a column.  
 **ALTER TABLE** Employee **DROP** email;

*The ALTER TABLE statement should be used with care and caution. You need to be careful to track and update any code that may depend on the existing table structure before you change it.*

# DROP Database The DROP DATABASE statement is used to delete a database.

**DROP DATABASE** Employee;  
  
Likewise in creating a database, trying to delete a database (or a table) that doesn’t exit, will result in an error unless you use IF EXISTS.

**DROP** DATABASE **IF EXISTS** Employee;

## Table The DROP TABLE statement is used to delete a table.

**DROP TABLE** Employee;

# Data Control Language (DCL)

It’s worth mentioning that there are other operations in SQL that are typically lumped under the idea of controlling the access to data stored in a database.  
This is where you can either “grant” or “revoke” permissions for people in the database. It’s two main statements are GRANT and REVOKE.

***Most of your time will be spent on manipulating data, some of the time defining the actual database itself, and a little bit of the time controlling who has access to different parts of it.***

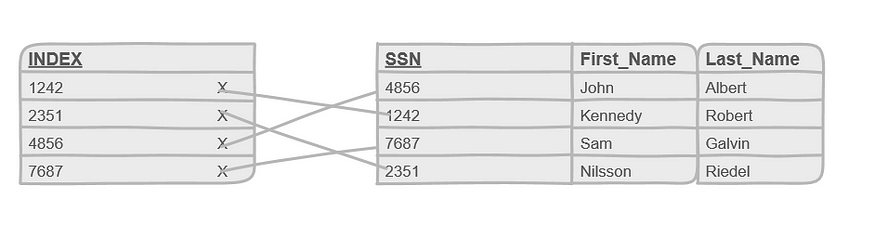
**Indexing**

One of the ways that will optimize your database searching and accessing is having indexes on the columns that you usually access the table using it.  
What the DBMS will do when you ask for a specific row, it will go sequentially and check with every row; “*Is this the row that I need?*”, If yes return it, if no, keep searching till the end.  
But, we have a better way to do that. An index, as we’ve mentioned, is a data structure, it won’t be obvious for you, but it’s stored inside the DBMS, most commonly as a B- tree.  
  
*By default, Most of the DBMS automatically create an index on primary and unique columns.*

**How Indexes Work?**

Let’s say that you have an index for a primary key. This will create an ordered list of primary key values in a separate table, each entry has a pointer points to the relative value in the original table.

So, whenever you want to access a table using the primary key, it will use binary search algorithm (takes time of O(LogN)) to access the required value in the Index table, and then, go to the relative value in the original table.



**Indexes**  
  
And, definitely, you can create another index on another column, even if it’s a non-primary column, like *first name*, assuming that you usually access the table using that column.  
  
The decision for choosing another column (besides the primary key) to be indexed-ed can be delayed until the database has been used for a while. This is because we want to know how users are really using our database, and what kind of queries they’re running rather than how we hoped or thought.