DBMS & SQL Tutorial for Beginners

DBMS Tutorial for Beginners is an amazing tutorial series to understand about Database Management System, its architecture and various techniques related to DBMS. In the SQL Tutorial, you will learn how to use SQL queries to fetch, insert, delete, update data in a Database.

Course Structure →

Database Concept

- Overview of DBMS
- Components of DBMS
- Database Architecture
- Types of Database Model
- ER Model: Basic Concepts
- ER Model: Creating ER Diagram
- The Extended ER Model
- Codd's 12 rule of RDBMS
- Basic Concepts of RDBMS
- Relational Algebra
- Relational Calculus
- ER Model to Relational Model
- Types of Database key
- Introduction to Normalization
- First Normal Form (1NF)
- Second Normal Form (1NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

Basic SQL

- SQL Intoduction
- Create query
- Alter query
- Truncate, Drop and Rename query
- All DML command
- All TCL Command
- All DCL Command
- WHERE clause
- SELECT query
- LIKE clause
- ORDER BY clause
- Group BY clause
- Having clause
- DISTINCT keyword
- AND & OR operator
- DIVISION operator

Advanced SQL

- SQL Constraints
- SQL function
- SQL Join
- SQL Alias
- SQL SET operation
- SQL Sequences
- SQL Views

What is Data?

Data is nothing but facts and statistics stored or free flowing over a network, generally it's raw and unprocessed. For example: When you visit any website, they might store you IP address, that is data, in return they might add a cookie in your browser, marking you that you visited the website, that is data, your name, it's data, your age, it's data.

Data becomes **information** when it is processed, turning it into something meaningful. Like, based on the cookie data saved on user's browser, if a website can analyse that generally men of age 20-25 visit us more, that is information, derived from the data collected.

What is a Database?

A **Database** is a collection of related data organised in a way that data can be easily accessed, managed and updated. Database can be software based or hardware based, with one sole purpose, storing data.

During early computer days, data was collected and stored on tapes, which were mostly write-only, which means once data is stored on it, it can never be read again. They were slow and bulky, and soon computer scientists realised that they needed a better solution to this problem.

Larry Ellison, the co-founder of **Oracle** was amongst the first few, who realised the need for a software based Database Management System.

What is DBMS?

A **DBMS** is a software that allows creation, definition and manipulation of database, allowing users to store, process and analyse data easily. DBMS provides us with an interface or a tool, to perform various operations like creating database, storing data in it, updating data, creating tables in the database and a lot more.

DBMS also provides protection and security to the databases. It also maintains data consistency in case of multiple users.

Here are some examples of popular DBMS used these days:

- MySql
- Oracle
- SQL Server
- IBM DB2
- PostgreSQL
- Amazon SimpleDB (cloud based) etc.

Characteristics of Database Management System

A database management system has following characteristics:

1. **Data stored into Tables:** Data is never directly stored into the database. Data is stored into tables, created inside the database. DBMS also allows to have relationships between tables which makes the data more meaningful and connected. You can easily understand what type of data is stored where by looking at all the tables created in a database.
2. **Reduced Redundancy:** In the modern world hard drives are very cheap, but earlier when hard drives were too expensive, unnecessary repetition of data in database was a big problem. But DBMS follows **Normalisation** which divides the data in such a way that repetition is minimum.
3. **Data Consistency:** On Live data, i.e. data that is being continuosly updated and added, maintaining the consistency of data can become a challenge. But DBMS handles it all by itself.

4. **Support Multiple user and Concurrent Access:** DBMS allows multiple users to work on it(update, insert, delete data) at the same time and still manages to maintain the data consistency.
5. **Query Language:** DBMS provides users with a simple Query language, using which data can be easily fetched, inserted, deleted and updated in a database.
6. **Security:** The DBMS also takes care of the security of data, protecting the data from un-authorised access. In a typical DBMS, we can create user accounts with different access permissions, using which we can easily secure our data by restricting user access.
7. DBMS supports **transactions**, which allows us to better handle and manage data integrity in real world applications where multi-threading is extensively used.

Advantages of DBMS

- Segregation of applicaion program.
- Minimal data duplicacy or data redundancy.
- Easy retrieval of data using the Query Language.
- Reduced development time and maintainance need.
- With Cloud Datacenters, we now have Database Management Systems capable of storing almost infinite data.
- Seamless integration into the application programming languages which makes it very easier to add a database to almost any application or website.

Disadvantages of DBMS

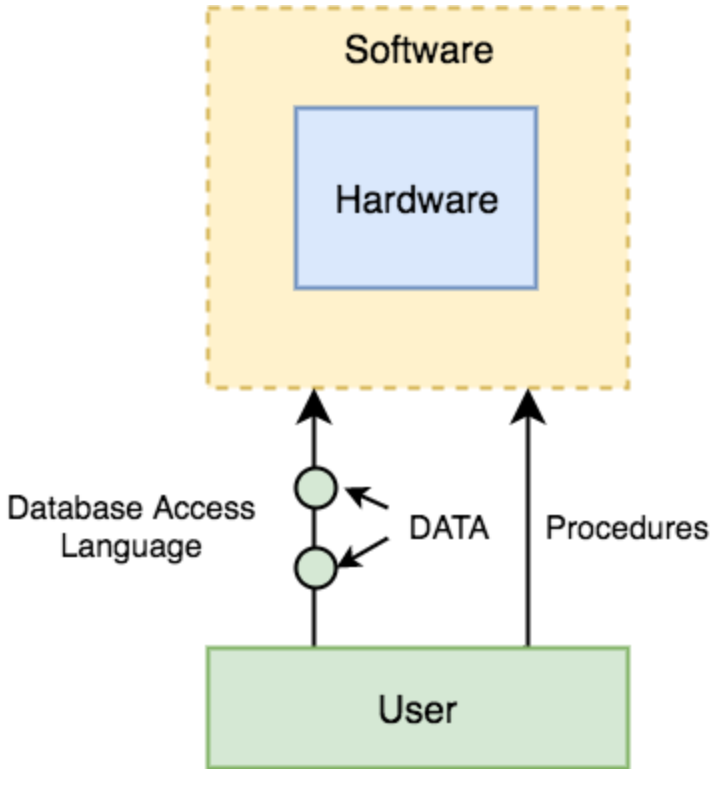
- It's Complexity
- Except MySQL, which is open source, licensed DBMSs are generally costly.
- They are large in size.

Components of DBMS

The database management system can be divided into five major components, they are:

1. Hardware
2. Software
3. Data
4. Procedures
5. Database Access Language

Let's have a simple diagram to see how they all fit together to form a database management system.



DBMS Components: Hardware

When we say Hardware, we mean computer, hard disks, I/O channels for data, and any other physical component involved before any data is successfully stored into the memory.

When we run Oracle or MySQL on our personal computer, then our computer's Hard Disk, our Keyboard using which we type in all the commands, our computer's RAM, ROM all become a part of the DBMS hardware.

DBMS Components: Software

This is the main component, as this is the program which controls everything. The DBMS software is more like a wrapper around the physical database, which provides us with an easy-to-use interface to store, access and update data.

The DBMS software is capable of understanding the Database Access Language and intrepret it into actual database commands to execute them on the DB.

DBMS Components: Data

Data is that resource, for which DBMS was designed. The motive behind the creation of DBMS was to store and utilise data.

In a typical Database, the user saved Data is present and **meta data** is stored.

Metadata is data about the data. This is information stored by the DBMS to better understand the data stored in it.

For example: When I store my **Name** in a database, the DBMS will store when the name was stored in the database, what is the size of the name, is it stored as related data to some other data, or is it independent, all this information is metadata.

DBMS Components: Procedures

Procedures refer to general instructions to use a database management system. This includes procedures to setup and install a DBMS, To login and logout of DBMS software, to manage databases, to take backups, generating reports etc.

DBMS Components: Database Access Language

Database Access Language is a simple language designed to write commands to access, insert, update and delete data stored in any database.

A user can write commands in the Database Access Language and submit it to the DBMS for execution, which is then translated and executed by the DBMS.

User can create new databases, tables, insert data, fetch stored data, update data and delete the data using the access language.

Users

- **Database Administrators:** Database Administrator or DBA is the one who manages the complete database management system. DBA takes care of the security of the DBMS, it's availability, managing the license keys, managing user accounts and access etc.
- **Application Programmer or Software Developer:** This user group is involved in developing and desiging the parts of DBMS.
- **End User:** These days all the modern applications, web or mobile, store user data. How do you think they do it? Yes, applications are programmed in such a way that they collect user data and store the data on DBMS systems running on their server. End users are the one who store, retrieve, update and delete data.

Understanding DBMS Architecture

A Database Management system is not always directly available for users and applications to access and store data in it. A Database Management system can be **centralised**(all the data stored at one location), **decentralised**(multiple copies of database at different locations) or **hierarchical**, depending upon its architecture.

1-tier DBMS architecture also exist, this is when the database is directly available to the user for using it to store data. Generally such a setup is used for local application development, where programmers communicate directly with the database for quick response.

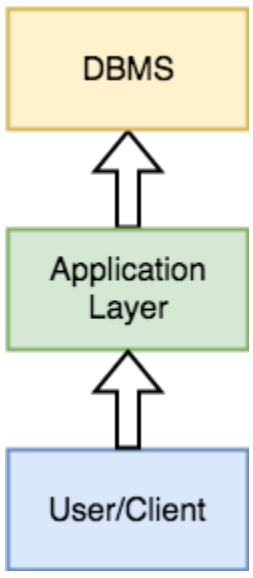
Database Architecture is logically of two types:

1. 2-tier DBMS architecture
2. 3-tier DBMS architecture

2-tier DBMS Architecture

2-tier DBMS architecture includes an **Application layer** between the user and the DBMS, which is responsible to communicate the user's request to the database management system and then send the response from the DBMS to the user.

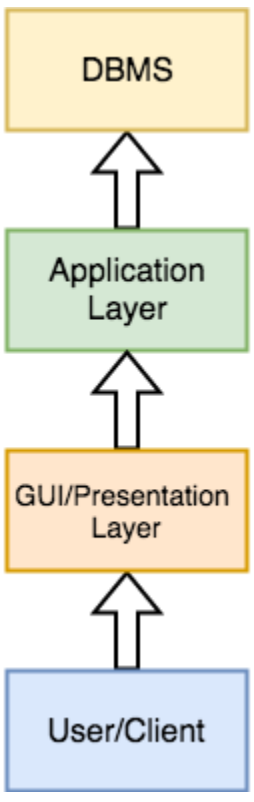
An application interface known as **ODBC**(Open Database Connectivity) provides an API that allow client side program to call the DBMS. Most DBMS vendors provide ODBC drivers for their DBMS.



Such an architecture provides the DBMS extra security as it is not exposed to the End User directly. Also, security can be improved by adding security and authentication checks in the Application layer too.

3-tier DBMS Architecture

3-tier DBMS architecture is the most commonly used architecture for web applications.



It is an extension of the 2-tier architecture. In the 2-tier architecture, we have an application layer which can be accessed programatically to perform various operations on the DBMS. The application generally understands the Database Access Language and processes end users requests to the DBMS.

In 3-tier architecture, an additional Presentation or GUI Layer is added, which provides a graphical user interface for the End user to interact with the DBMS.

For the end user, the GUI layer is the Database System, and the end user has no idea about the application layer and the DBMS system.

If you have used **MySQL**, then you must have seen **PHPMyAdmin**, it is the best example of a 3-tier DBMS architecture.

DBMS Database Models

A Database model defines the logical design and structure of a database and defines how data will be stored, accessed and updated in a database management system. While the **Relational Model** is the most widely used database model, there are other models too:

- Hierarchical Model
- Network Model
- Entity-relationship Model
- Relational Model

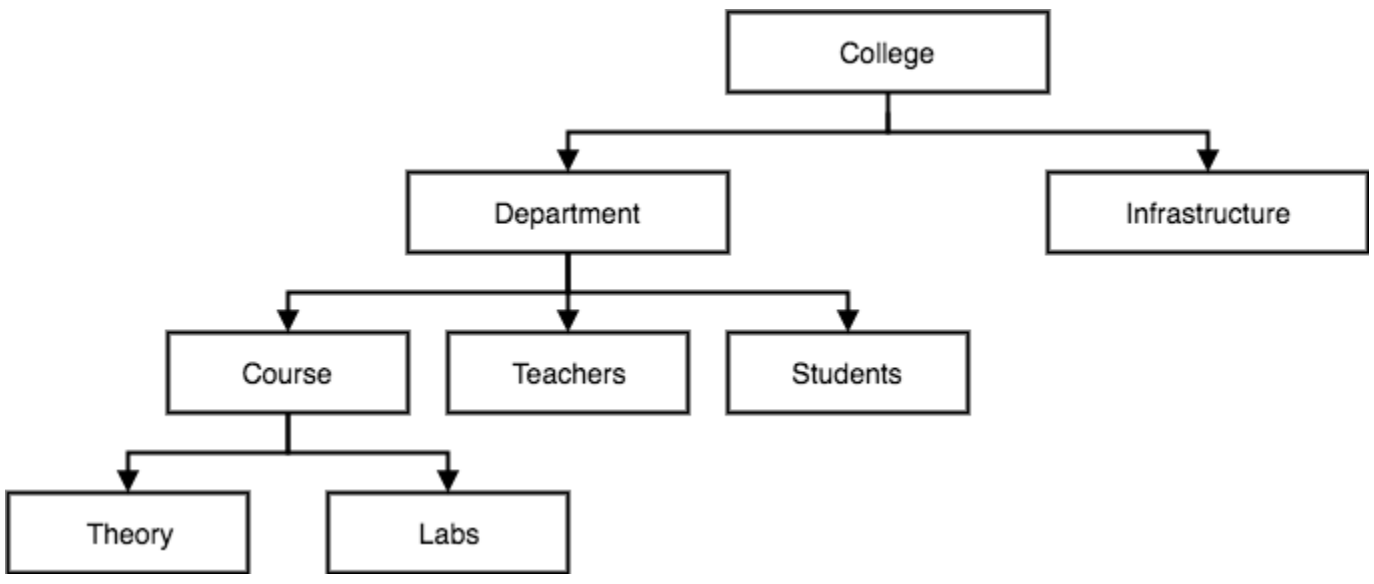
Hierarchical Model

This database model organises data into a tree-like-structure, with a single root, to which all the other data is linked. The heirarchy starts from the **Root** data, and expands like a tree, adding child nodes to the parent nodes.

In this model, a child node will only have a single parent node.

This model efficiently describes many real-world relationships like index of a book, recipes etc.

In hierarchical model, data is organised into tree-like structure with one one-to-many relationship between two different types of data, for example, one department can have many courses, many professors and of-course many students.

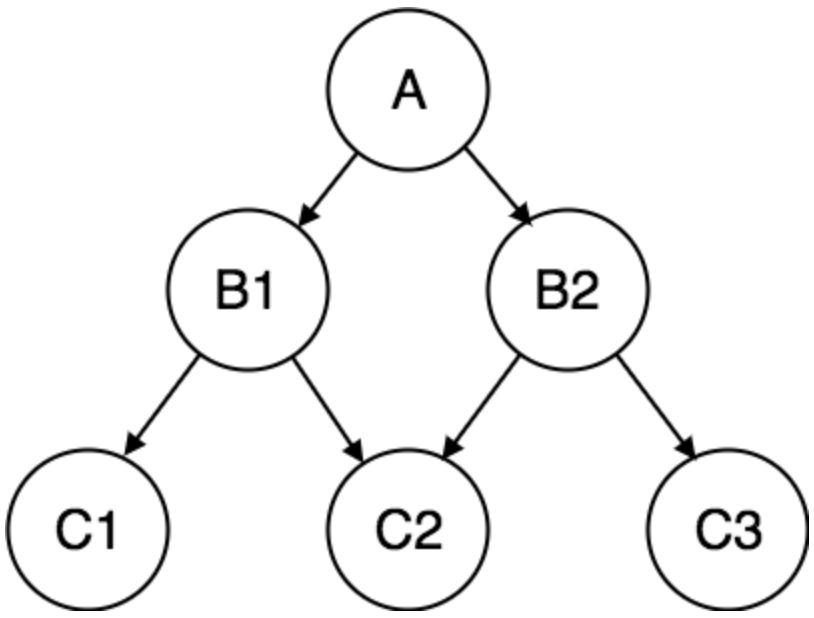


Network Model

This is an extension of the Hierarchical model. In this model data is organised more like a graph, and are allowed to have more than one parent node.

In this database model data is more related as more relationships are established in this database model. Also, as the data is more related, hence accessing the data is also easier and fast. This database model was used to map many-to-many data relationships.

This was the most widely used database model, before Relational Model was introduced.



Entity-relationship Model

In this database model, relationships are created by dividing object of interest into entity and its characteristics into attributes.

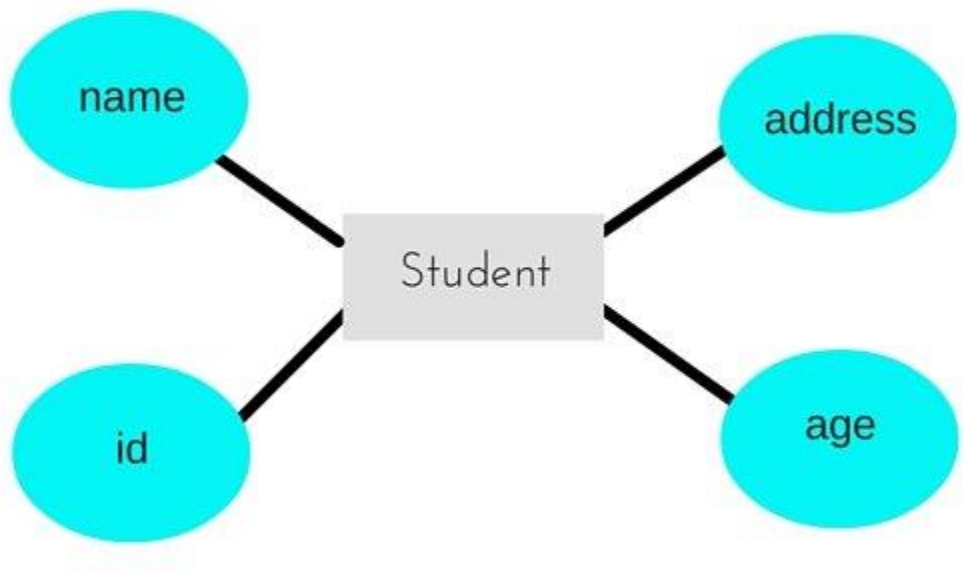
Different entities are related using relationships.

E-R Models are defined to represent the relationships into pictorial form to make it easier for different stakeholders to understand.

This model is good to design a database, which can then be turned into tables in relational model(explained below).

Let's take an example, If we have to design a School Database, then **Student** will be an **entity** with **attributes** name, age, address etc. As **Address** is generally complex, it can be another **entity** with **attributes** street name, pincode, city etc, and there will be a relationship between them.

Relationships can also be of different types. To learn about E-R Diagrams in details, click on the link.



Relational Model

In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

This model was introduced by E.F Codd in 1970, and since then it has been the most widely used database model, infact, we can say the only database model used around the world.

The basic structure of data in the relational model is tables. All the information related to a particular type is stored in rows of that table.

Hence, tables are also known as **relations** in relational model.

In the coming tutorials we will learn how to design tables, normalize them to reduce data redundancy and how to use Structured Query language to access data from tables.

student_id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_id	subject_id	marks
1	1	98
1	2	78
2	1	76
3	2	88

Basic Concepts of ER Model in DBMS

As we described in the tutorial Database models, Entity-relationship model is a model used for design and representation of relationships between data.

The main data objects are termed as Entities, with their details defined as attributes, some of these attributes are important and are used to identity the entity, and different entities are related using relationships.

In short, to understand about the ER Model, we must understand about:

- Entity and Entity Set
- What are Attributes? And Types of Attributes.
- Keys
- Relationships

Let's take an example to explain everything. For a **School Management Software**, we will have to store **Student** information, **Teacher** information, **Classes**, **Subjects** taught in each class etc.

ER Model: Entity and Entity Set

Considering the above example, **Student** is an entity, **Teacher** is an entity,
similarly, **Class**, **Subject** etc are also entities.

An Entity is generally a real-world object which has characteristics and holds relationships in a DBMS.

If a Student is an Entity, then the complete dataset of all the students will be the **Entity Set**

ER Model: Attributes

If a Student is an Entity, then student's **roll no.**, student's **name**, student's **age**, student's **gender** etc will be its attributes.

An attribute can be of many types, here are different types of attributes defined in ER database model:

1. **Simple attribute:** The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, student's **age**.
2. **Composite attribute:** A composite attribute is made up of more than one simple attribute. For example, student's **address** will contain, **house no.**, **street name**, **pincode** etc.
3. **Derived attribute:** These are the attributes which are not present in the whole database management system, but are derived using other attributes. For example, *average age of students in a class*.
4. **Single-valued attribute:** As the name suggests, they have a single value.
5. **Multi-valued attribute:** And, they can have multiple values.

ER Model: Keys

If the attribute **roll no.** can uniquely identify a student entity, amongst all the students, then the attribute **roll no.** will be said to be a key.

Following are the types of Keys:

1. Super Key
2. Candidate Key
3. Primary Key

We have covered Keys in details here in Database Keys tutorial.

ER Model: Relationships

When an Entity is related to another Entity, they are said to have a relationship. For example, A **Class** Entity is related to **Student** entity, becasue students study in classes, hence this is a relationship.

Depending upon the number of entities involved, a **degree** is assigned to relationships.

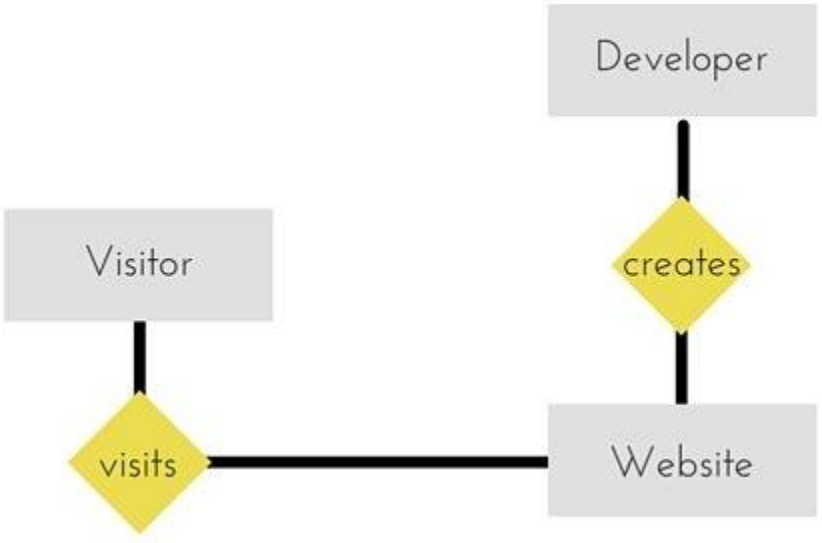
For example, if 2 entities are involved, it is said to be **Binary relationship**, if 3 entities are involved, it is said to be **Ternary** relationship, and so on.

In the next tutorial, we will learn how to create ER diagrams and design databases using ER diagrams.

Working with ER Diagrams

ER Diagram is a visual representation of data that describes how data is related to each other. In ER Model, we disintegrate data into entities, attributes and setup relationships between entities, all this can be represented visually using the ER diagram.

For example, in the below diagram, anyone can see and understand what the diagram wants to convey: *Developer develops a website, whereas a Visitor visits a website*.



Components of ER Diagram

Entitiy, Attributes, Relationships etc form the components of ER Diagram and there are defined symbols and shapes to represent each one of them.

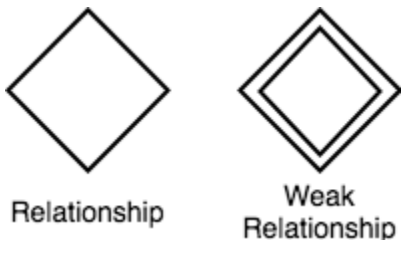
Let's see how we can represent these in our ER Diagram.

Entity

Simple rectangular box represents an Entity.

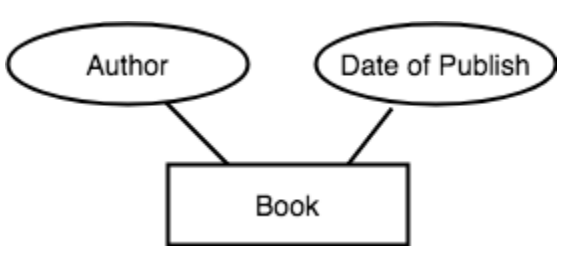
Relationships between Entities - Weak and Strong

Rhombus is used to setup relationships between two or more entities.



Attributes for any Entity

Ellipse is used to represent attributes of any entity. It is connected to the entity.

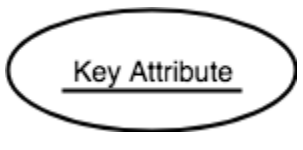


Weak Entity

A weak Entity is represented using double rectangular boxes. It is generally connected to another entity.

Key Attribute for any Entity

To represent a Key attribute, the attribute name inside the Ellipse is underlined.



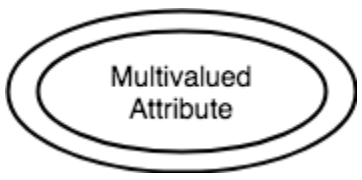
Derived Attribute for any Entity

Derived attributes are those which are derived based on other attributes, for example, age can be derived from date of birth.

To represent a derived attribute, another dotted ellipse is created inside the main ellipse.

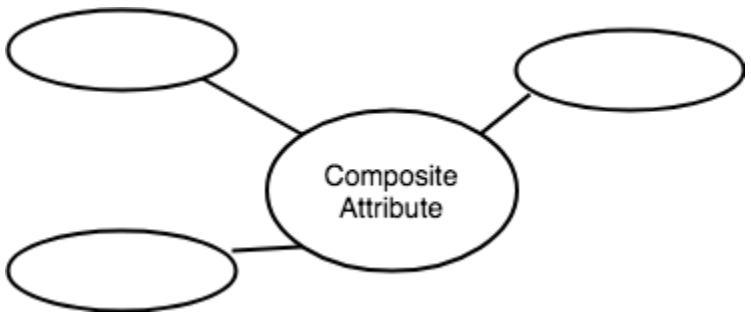
Multivalued Attribute for any Entity

Double Ellipse, one inside another, represents the attribute which can have multiple values.



Composite Attribute for any Entity

A composite attribute is the attribute, which also has attributes.



ER Diagram: Entity

An **Entity** can be any object, place, person or class. In ER Diagram, an **entity** is represented using rectangles. Consider an example of an Organisation- Employee, Manager, Department, Product and many more can be taken as entities in an Organisation.

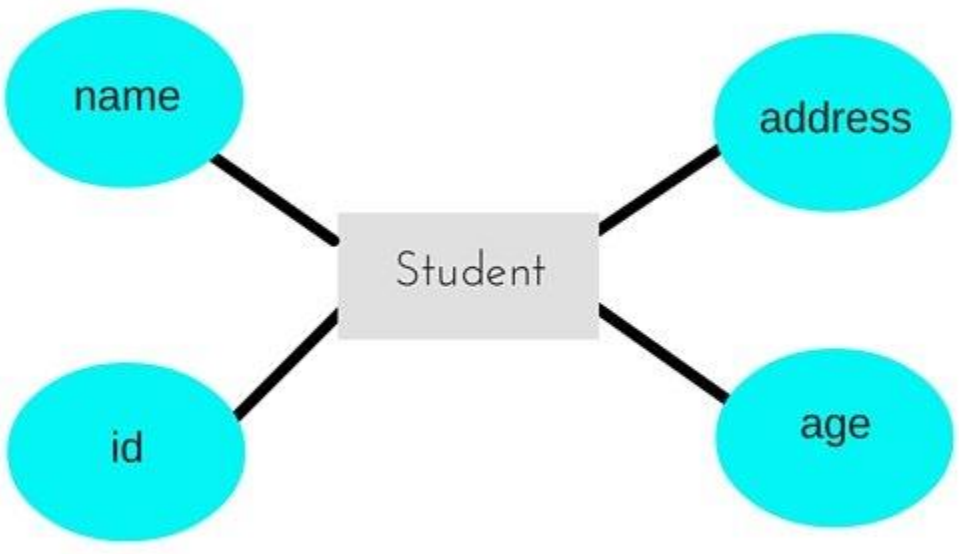
The yellow rhombus in between represents a relationship.

ER Diagram: Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doesn't have anay key attribute of its own. Double rectangle is used to represent a weak entity.

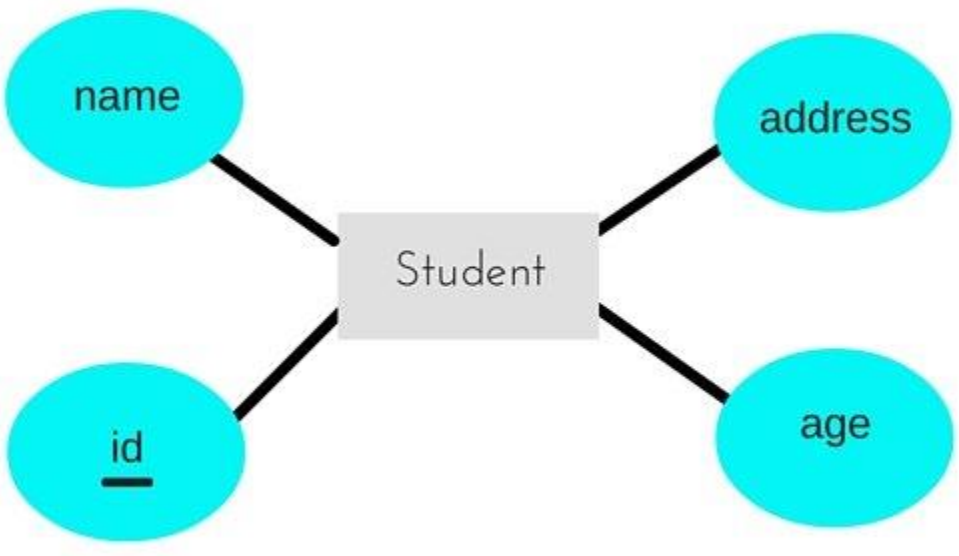
ER Diagram: Attribute

An **Attribute** describes a property or characterstic of an entity. For example, **Name**, **Age**, **Address** etc can be attributes of a **Student**. An attribute is represented using eclipse.



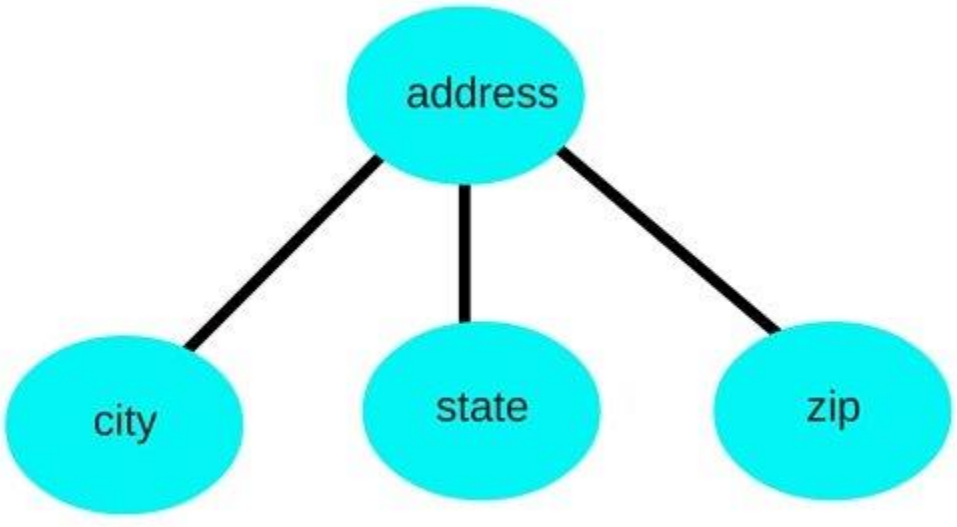
ER Diagram: Key Attribute

Key attribute represents the main characterstic of an Entity. It is used to represent a Primary key. Ellipse with the text underlined, represents Key Attribute.



ER Diagram: Composite Attribute

An attribute can also have their own attributes. These attributes are known as **Composite** attributes.



ER Diagram: Relationship

A Relationship describes relation between **entities**. Relationship is represented using diamonds or rhombus.

There are three types of relationship that exist between Entities.

1. Binary Relationship
2. Recursive Relationship
3. Ternary Relationship

ER Diagram: Binary Relationship

Binary Relationship means relation between two Entities. This is further divided into three types.

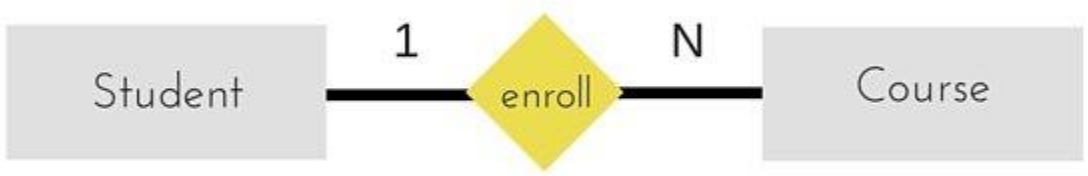
One to One Relationship

This type of relationship is rarely seen in real world.

The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in real-world relationships.

One to Many Relationship

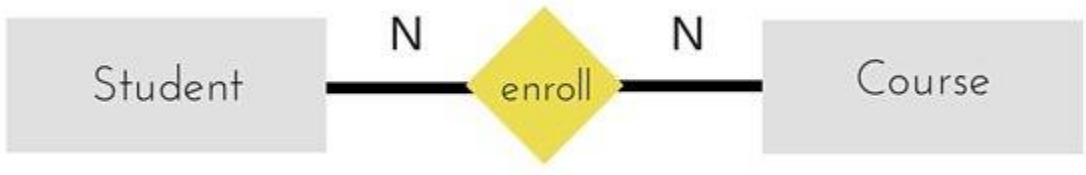
The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.



Many to One Relationship

It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.

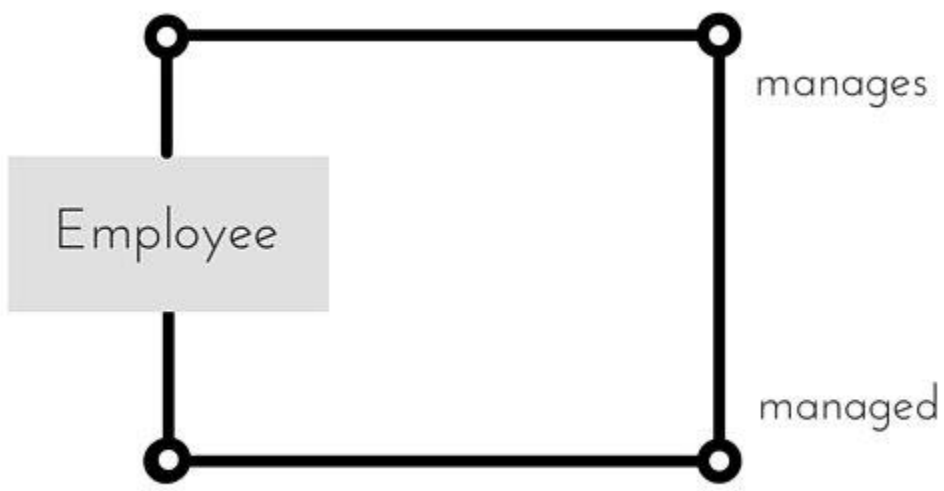
Many to Many Relationship



The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.

ER Diagram: Recursive Relationship

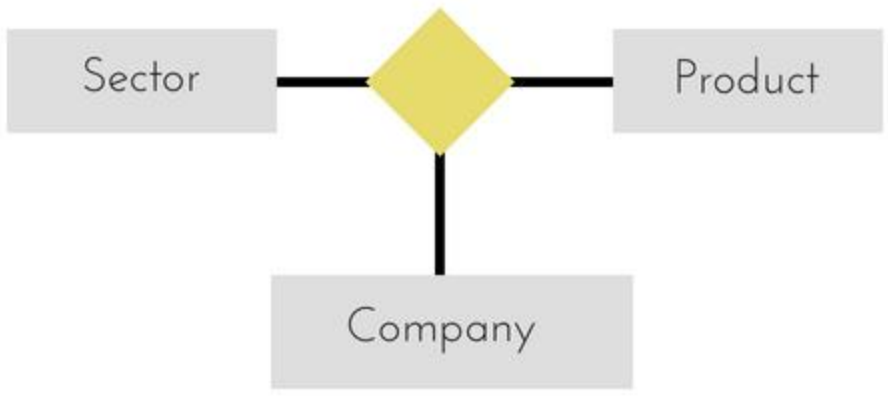
When an Entity is related with itself it is known as **Recursive** Relationship.



ER Diagram: Ternary Relationship

Relationship of degree three is called Ternary relationship.

A Ternary relationship involves three entities. In such relationships we always consider two entites together and then look upon the third.



- The above relationship involves 3 entities.
- Company operates in Sector, producing some Products.

For example, in the diagram above, we have three related entities, **Company**, **Product** and **Sector**. To understand the relationship better or to define rules around the model, we should relate two entities and then derive the third one.

A **Company** produces many **Products**/ each product is produced by exactly one company.

A **Company** operates in only one **Sector** / each sector has many companies operating in it.

Considering the above two rules or relationships, we see that although the complete relationship involves three entities, but we are looking at two entities at a time.

The Enhanced ER Model

As the complexity of data increased in the late 1980s, it became more and more difficult to use the traditional ER Model for database modelling. Hence some improvements or enhancements were made to the existing ER Model to make it able to handle the complex applications better.

Hence, as part of the **Enhanced ER Model**, along with other improvements, three new concepts were added to the existing ER Model, they were:

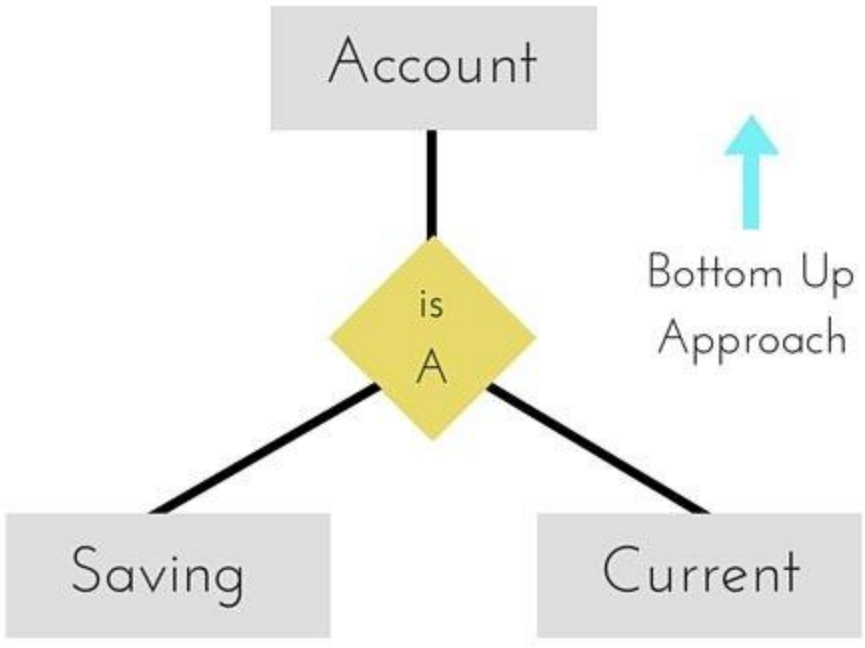
1. Generalization
2. Specialization
3. Aggregration

Let's understand what they are, and why were they added to the existing ER Model.

Generalization

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entities to make further higher level entity.

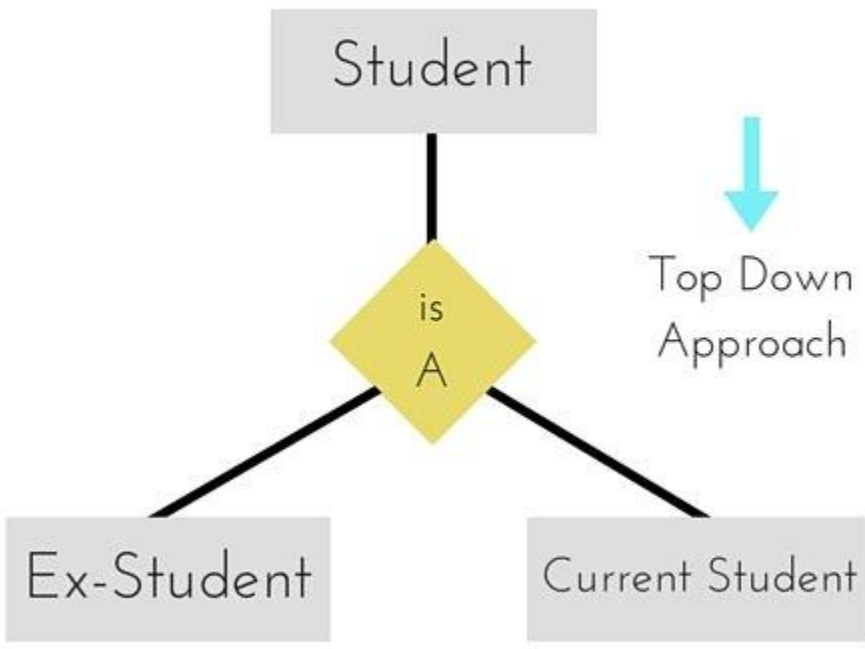
It's more like Superclass and Subclass system, but the only difference is the approach, which is bottom-up. Hence, entities are combined to form a more generalised entity, in other words, sub-classes are combined to form a super-class.



For example, **Saving** and **Current** account types entities can be generalised and an entity with name **Account** can be created, which covers both.

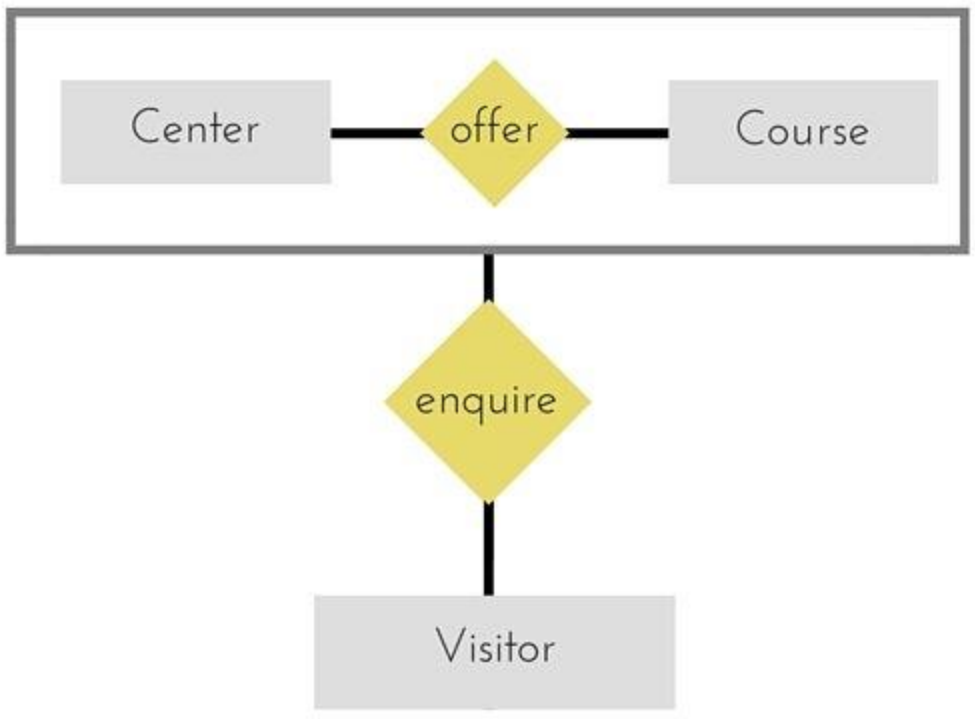
Specialization

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, a higher level entity may not have any lower-level entity sets, it's possible.



Aggregration

Aggregration is a process when relation between two entities is treated as a **single entity**.



In the diagram above, the relationship between **Center** and **Course** together, is acting as an Entity, which is in relationship with another entity **Visitor**. Now in real world, if a Visitor or a Student visits a Coaching Center, he/she will never enquire about the center only or just about the course, rather he/she will ask enquire about both.

Codd's Rule for Relational DBMS

E.F Codd was a Computer Scientist who invented the **Relational model** for Database management. Based on relational model, the **Relational database** was created. Codd proposed 13 rules popularly known as **Codd's 12 rules** to test DBMS's concept against his relational model. Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS). Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even **Oracle** follows only eight and half(8.5) out of 13. The Codd's 12 rules are as follows.

Rule zero

This rule states that for a system to qualify as an **RDBMS**, it must be able to manage database entirely through the relational capabilities.

Rule 1: Information rule

All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

Rule 2: Guaranted Access

Each unique piece of data(atomic value) should be accesible by : **Table Name + Primary Key(Row) + Attribute(column)**.

NOTE: Ability to directly access via POINTER is a violation of this rule.

Rule 3: Systematic treatment of NULL

`Null` has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Also, Primary key must not be null, ever. Expression on `NULL` must give null.

Rule 4: Active Online Catalog

Database dictionary(catalog) is the structure description of the complete **Database** and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

Rule 5: Powerful and Well-Structured Language

One well structured language must be there to provide all manners of access to the data stored in the database. Example: **SQL**, etc. If the database allows access to the data without the use of this language, then that is a violation.

Rule 6: View Updation Rule

All the view that are theoretically updatable should be updatable by the system as well.

Rule 7: Relational Level Operation

There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

Rule 8: Physical Data Independence

The physical storage of data should not matter to the system. If say, some file supporting table is renamed or moved from one disk to another, it should not effect the application.

Rule 9: Logical Data Independence

If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

Rule 10: Integrity Independence

The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc, should be stored in Data Dictionary. This also make **RDBMS** independent of front-end.

Rule 11: Distribution Independence

A database should work properly regardless of its distribution across a network. Even if a database is geographically distributed, with data stored in pieces, the end user should get an impression that it is stored at the same place. This lays the foundation of **distributed database**.

Rule 12: Nonsubversion Rule

If low level access is allowed to a system it should not be able to subvert or bypass integrity rules to change the data. This can be achieved by some sort of looking or encryption.

Basic Relational DBMS Concepts

A **Relational Database management System**(RDBMS) is a database management system based on the relational model introduced by E.F Codd. In relational model, data is stored in **relations**(tables) and is represented in form of **tuples**(rows).

RDBMS is used to manage Relational database. **Relational database** is a collection of organized set of tables related to each other, and from which data can be accessed easily. Relational Database is the most commonly used database these days.

RDBMS: What is Table ?

In Relational database model, a **table** is a collection of data elements organised in terms of rows and columns. A table is also considered as a convenient representation of **relations**. But a table can have duplicate row of data while a true **relation** cannot have duplicate data. Table is the most simplest form of data storage. Below is an example of an Employee table.

ID	Name	Age	Salary
1	Adam	34	13000
2	Alex	28	15000
3	Stuart	20	18000
4	Ross	42	19020

RDBMS: What is a Tuple?

A single entry in a table is called a **Tuple** or **Record** or **Row**. A **tuple** in a table represents a set of related data. For example, the above **Employee** table has 4 tuples/records/rows.

Following is an example of single record or tuple.

1	Adam	34	13000

RDBMS: What is an Attribute?

A table consists of several records(row), each record can be broken down into several smaller parts of data known as **Attributes**. The above **Employee** table consist of four attributes, **ID**, **Name**, **Age** and **Salary**.

Attribute Domain

When an attribute is defined in a relation(table), it is defined to hold only a certain type of values, which is known as **Attribute Domain**.

Hence, the attribute **Name** will hold the name of employee for every tuple. If we save employee's address there, it will be violation of the Relational database model.

Name
Adam
Alex
Stuart - 9/401, OC Street, Amsterdam
Ross

What is a Relation Schema?

A relation schema describes the structure of the relation, with the name of the relation(name of table), its attributes and their names and type.

What is a Relation Key?

A relation key is an attribute which can uniquely identify a particular tuple(row) in a relation(table).

Relational Integrity Constraints

Every relation in a relational database model should abide by or follow a few constraints to be a valid relation, these constraints are called as **Relational Integrity Constraints**.

The three main Integrity Constraints are:

1. Key Constraints
2. Domain Constraints
3. Referential integrity Constraints

Key Constraints
We store data in tables, to later access it whenever required. In every table one or more than one attributes together are used to fetch data from tables. The **Key Constraint** specifies that there should be such an attribute(column) in a relation(table), which can be used to fetch data for any tuple(row).

The Key attribute should never be **NULL** or same for two different row of data.

For example, in the **Employee** table we can use the attribute `ID` to fetch data for each of the employee. No value of `ID` is null and it is unique for every row, hence it can be our **Key attribute**.

Domain Constraint
Domain constraints refers to the rules defined for the values that can be stored for a certain attribute.

Like we explained above, we cannot store **Address** of employee in the column for **Name**.

Similarly, a mobile number cannot exceed 10 digits.

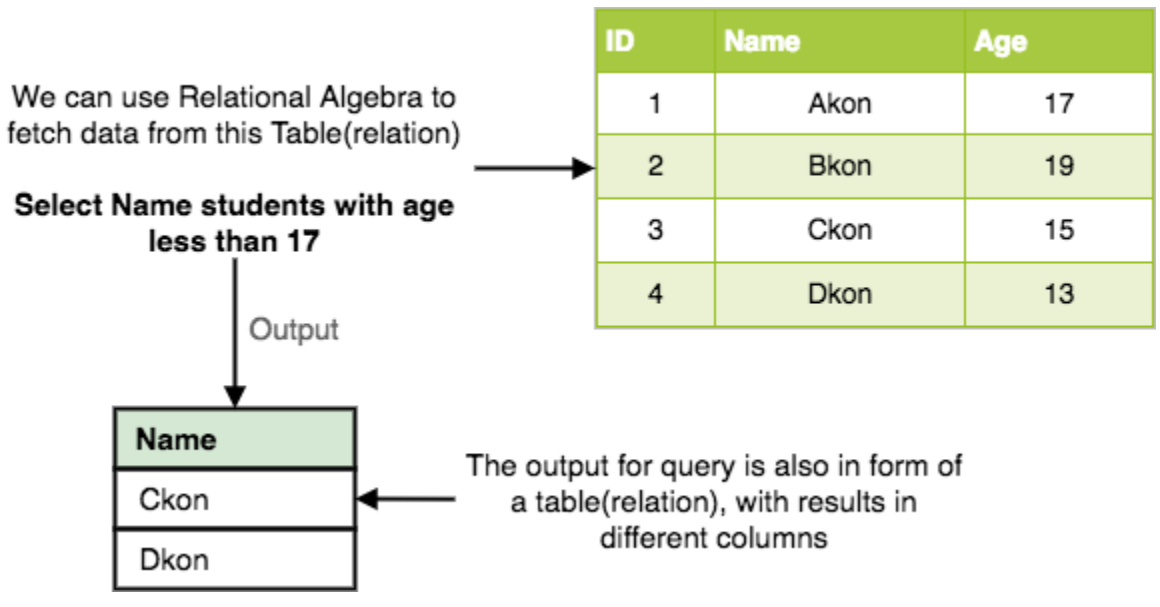
Referential Integrity Constraint
We will study about this in detail later. For now remember this example, if I say **Supriya** is my girlfriend, then a girl with name Supriya should also exist for that relationship to be present.

If a table reference to some data from another table, then that table and that data should be present for referential integrity constraint to hold true.

What is Relational Algebra?

Every database management system must define a query language to allow users to access the data stored in the database. **Relational Algebra** is a procedural query language used to query the database tables to access data in different ways.

In relational algebra, input is a relation(table from which data has to be accessed) and output is also a relation(a temporary table holding the data asked for by the user).



Relational Algebra works on the whole table at once, so we do not have to use loops etc to iterate over all the rows(tuples) of data one by one. All we have to do is specify the table name from

which we need the data, and in a single line of command, relational algebra will traverse the entire given table to fetch data for you.

The primary operations that we can perform using relational algebra are:

1. Select
2. Project
3. Union
4. Set Different
5. Cartesian product
6. Rename

Select Operation (σ)

This is used to fetch rows(tuples) from table(relation) which satisfies a given condition.

Syntax: $\sigma_p(r)$

Where, σ represents the Select Predicate, r is the name of relation(table name in which you want to look for data), and p is the prepositional logic, where we specify the conditions that must be satisfied by the data. In prepositional logic, one can use **unary** and **binary** operators like $=$, $<$, $>$ etc, to specify the conditions.

Let's take an example of the Student table we specified above in the Introduction of relational algebra, and fetch data for **students** with **age** more than 17.

$\sigma_{age > 17}(Student)$

This will fetch the tuples(rows) from table **Student**, for which **age** will be greater than **17**.

You can also use, and, or etc operators, to specify two conditions, for example,

$\sigma_{age > 17 \text{ and } gender = \text{'Male'}}(Student)$

This will return tuples(rows) from table **Student** with information of male students, of age more than 17.(Consider the Student table has an attribute $Gender$ too.)

Project Operation (∏)

Project operation is used to project only a certain set of attributes of a relation. In simple words, If you want to see only the **names** all of the students in the **Student** table, then you can use Project Operation.

It will only project or show the columns or attributes asked for, and will also remove duplicate data from the columns.

Syntax: $\prod_{A1, A2...}(r)$

where A1, A2 etc are attribute names(column names).

For example,

$\prod_{Name, Age}(Student)$

Above statement will show us only the **Name** and **Age** columns for all the rows of data in **Student** table.

Union Operation (∪)

This operation is used to fetch data from two relations(tables) or temporary relation(result of another operation).

For this operation to work, the relations(tables) specified should have same number of attributes(columns) and same attribute domain. Also the duplicate tuples are autamatically eliminated from the result.

Syntax: $A \cup B$

where A and B are relations.

For example, if we have two tables **RegularClass** and **ExtraClass**, both have a column **student** to save name of student, then,

$\prod_{Student}(RegularClass) \cup \prod_{Student}(ExtraClass)$

Above operation will give us name of **Students** who are attending both regular classes and extra classes, eliminating repetition.

Set Difference (-)

This operation is used to find data present in one relation and not present in the second relation. This operation is also applicable on two relations, just like Union operation.

Syntax: $A - B$

where A and B are relations.

For example, if we want to find name of students who attend the regular class but not the extra class, then, we can use the below operation:

$\prod_{Student}(RegularClass) - \prod_{Student}(ExtraClass)$

Cartesian Product (X)

This is used to combine data from two different relations(tables) into one and fetch data from the combined relation.

Syntax: $A \times B$

For example, if we want to find the information for Regular Class and Extra Class which are conducted during morning, then, we can use the following operation:

$\sigma_{time = 'morning'} (RegularClass \times ExtraClass)$

For the above query to work, both **RegularClass** and **ExtraClass** should have the attribute **time**.

Rename Operation (ρ)

This operation is used to rename the output relation for any query operation which returns result like Select, Project etc. Or to simply rename a relation(table)

Syntax: `ρ(RelationNew, RelationOld)`

Apart from these common operations Relational Algebra is also used for **Join** operations like,

- Natural Join
- Outer Join
- Theta join etc.

What is Relational Calculus?

Contrary to Relational Algebra which is a procedural query language to fetch data and which also explains how it is done, **Relational Calculus** in non-procedural query language and has no description about how the query will work or the data will b fetched. It only focusses on what to do, and not on how to do it.

Relational Calculus exists in two forms:

1. Tuple Relational Calculus (TRC)
2. Domain Relational Calculus (DRC)

Tuple Relational Calculus (TRC)

In tuple relational calculus, we work on filtering tuples based on the given condition.

Syntax: `{ T | Condition }`

In this form of relational calculus, we define a tuple variable, specify the table(relation) name in which the tuple is to be searched for, along with a condition.

We can also specify column name using a `.` dot operator, with the tuple variable to only get a certain attribute(column) in result.

A lot of informtion, right! Give it some time to sink in.

A tuple variable is nothing but a name, can be anything, generally we use a single alphabet for this, so let's say `T` is a tuple variable.

To specify the name of the relation(table) in which we want to look for data, we do the following:

`Relation(T)`, where `T` is our tuple variable.

For example if our table is **Student**, we would put it as `Student(T)`

Then comes the condition part, to specify a condition applicable for a particluar attribute(column), we can use the `.` dot variable with the tuple variable to specify it, like in table **Student**, if we want to get data for students with age greater than 17, then, we can write it as,

`T.age > 17`, where `T` is our tuple variable.

Putting it all together, if we want to use Tuple Relational Calculus to fetch names of students, from table **Student**, with age greater than **17**, then, for `T` being our tuple variable,

```
T.name | Student(T) AND T.age > 17
```

Domain Relational Calculus (DRC)

In domain relational calculus, filtering is done based on the domain of the attributes and not based on the tuple values.

Syntax: `{ c1, c2, c3, ..., cn | F(c1, c2, c3, ... ,cn)}`

where, c1, c2... etc represents domain of attributes(columns) and `F` defines the formula including the condition for fetching the data.

For example,

```
{< name, age > | ∈ Student ∧ age > 17}
```

Again, the above query will return the names and ages of the students in the table **Student** who are older than 17.

ER Model to Relational Model

As we all know that ER Model can be represented using ER Diagrams which is a great way of designing and representing the database design in more of a flow chart form.

It is very convenient to design the database using the ER Model by creating an ER diagram and later on converting it into relational model to design your tables.

Not all the ER Model constraints and components can be directly transformed into relational model, but an approximate schema can be derived.

So let's take a few examples of ER diagrams and convert it into relational model schema, hence creating tables in RDBMS.

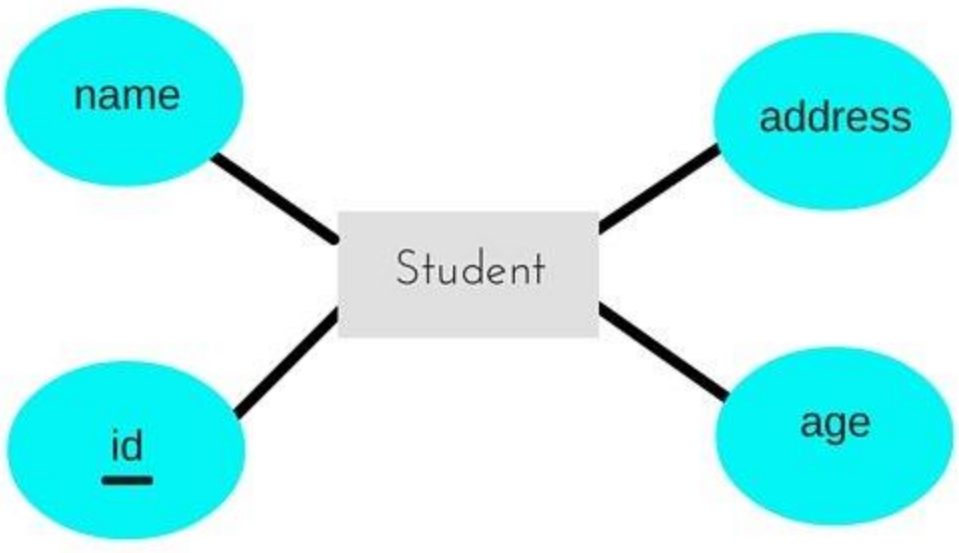
Entity becomes Table

Entity in ER Model is changed into tables, or we can say for every Entity in ER model, a table is created in Relational Model.

And the **attributes** of the Entity gets converted to columns of the table.

And the primary key specified for the entity in the ER model, will become the primary key for the table in relational model.

For example, for the below ER Diagram in ER Model,



A table with name **Student** will be created in relational model, which will have 4 columns, id, name, age, address and id will be the primary key for this table.

Relationship becomes a Relationship Table

In ER diagram, we use diamond/rhombus to reprsent a relationship between two entities. In Relational model we create a relationship table for ER Model relationships too.

In the ER diagram below, we have two entities **Teacher** and **Student** with a relationship between them.

As discussd above, entity gets mapped to table, hence we will create table for **Teacher** and a table for **Student** with all the attributes converted into columns.

Now, an additional table will be created for the relationship, for example **StudentTeacher** or give it any name you like. This table will hold the primary key for both Student and Teacher, in a tuple to describe the relationship, which teacher teaches which student.

If there are additional attributes related to this relationship, then they become the columns for this table, like subject name.

Also proper foriegn key constraints must be set for all the tables.

Points to Remember

Similarly we can generate relational database schema using the ER diagram. Following are some key points to keep in mind while doing so:

1. Entity gets converted into Table, with all the attributes becoming fields(columns) in the table.
2. Relationship between entities is also converted into table with primary keys of the related entities also stored in it as foreign keys.
3. Primary Keys should be properly set.
4. For any relationship of Weak Entity, if primary key of any other entity is included in a table, foriegn key constraint must be defined.

Introduction to Database Keys

Keys are very important part of Relational database model. They are used to establish and identify relationships between tables and also to uniquely identify any record or row of data inside a table.

A Key can be a single attribute or a group of attributes, where the combination may act as a key.

The video below covers all about the different keys in an RDBMS.

Why we need a Key?

In real world applications, number of tables required for storing the data is huge, and the different tables are related to each other as well.

Also, tables store a lot of data in them. Tables generally extends to thousands of records stored in them, unsorted and unorganised.

Now to fetch any particular record from such dataset, you will have to apply some conditions, but what if there is duplicate data present and every time you try to fetch some data by applying certain condition, you get the wrong data. How many trials before you get the right data?

To avoid all this, **Keys** are defined to easily identify any row of data in a table.

Let's try to understand about all the keys using a simple example.

student_id	name	phone	age
1	Akon	9876723452	17
2	Akon	9991165674	19
3	Bkon	7898756543	18
4	Ckon	8987867898	19
5	Dkon	9990080080	17

Let's take a simple **Student** table, with fields `student_id`, `name`, `phone` and `age`.

Super Key

Super Key is defined as a set of attributes within a table that can uniquely identify each record within a table. Super Key is a superset of Candidate key.

In the table defined above super key would include `student_id`, `(student_id, name)`, `phone` etc.

Confused? The first one is pretty simple as `student_id` is unique for every row of data, hence it can be used to identity each row uniquely.

Next comes, `(student_id, name)`, now name of two students can be same, but their `student_id` can't be same hence this combination can also be a key.

Similarly, phone number for every student will be unique, hence again, `phone` can also be a key.

So they all are super keys.

Candidate Key

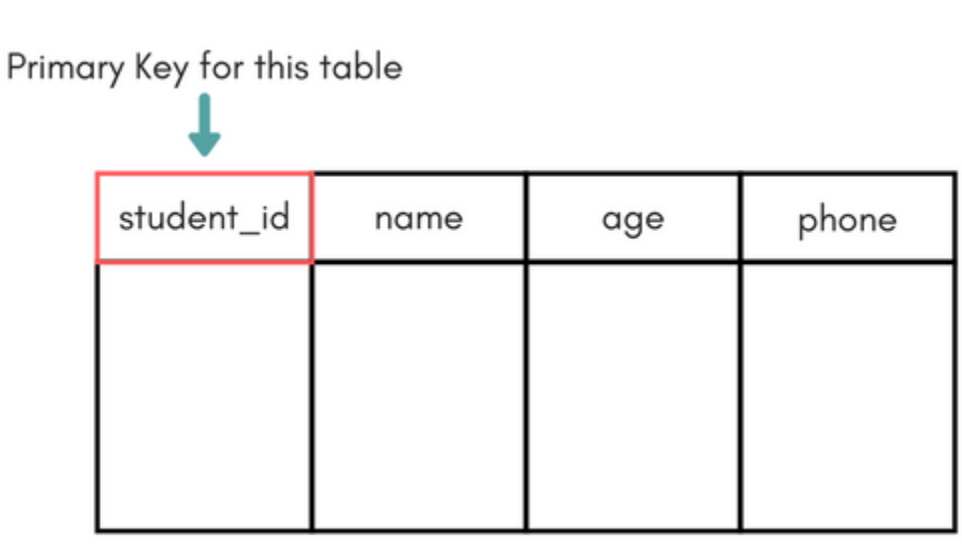
Candidate keys are defined as the minimal set of fields which can uniquely identify each record in a table. It is an attribute or a set of attributes that can act as a Primary Key for a table to uniquely identify each record in that table. There can be more than one candidate key.

In our example, `student_id` and `phone` both are candidate keys for table **Student**.

- A candiate key can never be NULL or empty. And its value should be unique.
- There can be more than one candidate keys for a table.
- A candidate key can be a combination of more than one columns(attributes).

Primary Key

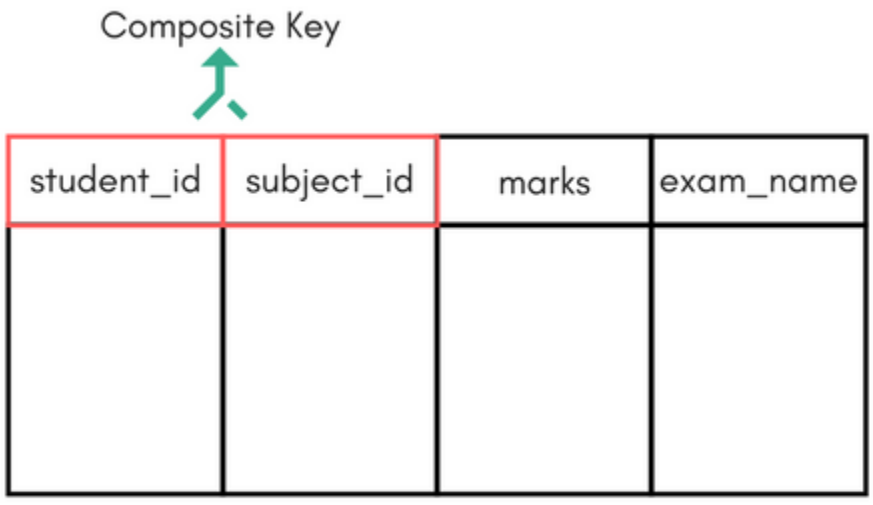
Primary key is a candidate key that is most appropriate to become the main key for any table. It is a key that can uniquely identify each record in a table.



For the table **Student** we can make the `student_id` column as the primary key.

Composite Key

Key that consists of two or more attributes that uniquely identify any record in a table is called **Composite key**. But the attributes which together form the **Composite key** are not a key independentely or individually.



Score Table - To save scores of the student for various subjects.

In the above picture we have a **Score** table which stores the marks scored by a student in a particular subject.

In this table `student_id` and `subject_id` together will form the primary key, hence it is a composite key.

Secondary or Alternative key

The candidate key which are not selected as primary key are known as secondary keys or alternative keys.

Non-key Attributes

Non-key attributes are the attributes or fields of a table, other than **candidate key** attributes/fields in a table.

Non-prime Attributes

Non-prime Attributes are attributes other than **Primary Key attribute(s).**.

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy(repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

- Eliminating redundant(useless) data.

- Ensuring data dependencies make sense i.e data is logically stored.

The **video** below will give you a good overview of Database Normalization. If you want you can skip the video, as the concept is covered in detail, below the video.

Problems Without Normalization

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a **Student** table.

rollno	name	branch	hod	office_tel
401	Akon	CSE	Mr. X	53337
402	Bkon	CSE	Mr. X	53337
403	Ckon	CSE	Mr. X	53337
404	Dkon	CSE	Mr. X	53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields `branch`, `hod`(Head of Department) and `office_tel` is repeated for the students who are in the same branch in the college, this is **Data Redundancy**.

Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as **NULL**.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but **Insertion anomalies**.

Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

In our **Student** table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

Normalization Rule

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form

First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

In the next tutorial, we will discuss about the **First Normal Form** in details.

Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

To understand what is Partial Dependency and how to normalize a table to 2nd normal for, jump to the **Second Normal Form** tutorial.

Third Normal Form (3NF)

A table is said to be in the Third Normal Form when,

1. It is in the Second Normal form.
2. And, it doesn't have Transitive Dependency.

Here is the **Third Normal Form** tutorial. But we suggest you to first study about the second normal form and then head over to the third normal form.

Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

- R must be in 3rd Normal Form
- and, for each functional dependency (X → Y), X should be a super Key.

To learn about BCNF in detail with a very easy to understand example, head to **Boye-Codd Normal Form** tutorial.

Fourth Normal Form (4NF)

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

Here is the **Fourth Normal Form** tutorial. But we suggest you to understand other normal forms before you head over to the fourth normal form.

What is First Normal Form (1NF)?

In this tutorial we will learn about the 1st(First) Normal Form which is more like the Step 1 of the Normalization process. The 1st Normal form expects you to design your table in such a way that it can easily be extended and it is easier for you to retrieve data from it whenever required.

In our last tutorial we learned and understood how data redundancy or repetition can lead to several issues like Insertion, Deletion and Updation anomalies and how **Normalization** can reduce data redundancy and make the data more meaningful.

If tables in a database are not even in the 1st Normal Form, it is considered as **bad database design**.

Rules for First Normal Form

The first normal form expects you to follow a few simple rules while designing your database, and they are:

Rule 1: Single Valued Attributes
Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now.

Rule 2: Attribute Domain should not change
This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

For example: If you have a column `dob` to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows.

Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.

If one or more columns have same name, then the DBMS system will be left confused.

Rule 4: Order doesn't matters

This rule says that the order in which you store the data in your table doesn't matter.

Time for an Example

Although all the rules are self explanatory still let's take an example where we will create a table to store student data which will have student's roll no., their name and the name of subjects they have opted for.

Here is our table, with some sample data added to it.

roll_no	name	subject
101	Akon	OS, CN
103	Ckon	Java
102	Bkon	C, C++

Our table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But out of the 3 different students in our table, 2 have opted for more than 1 subject. And we have stored the subject names in a single column. But as per the 1st Normal form each column must contain atomic value.

How to solve this Problem?

It's very simple, because all we have to do is break the values into atomic values.

Here is our updated table and it now satisfies the First Normal Form.

roll_no	name	subject

101	Akon	OS
101	Akon	CN
103	Ckon	Java
102	Bkon	C
102	Bkon	C++

By doing so, although a few values are getting repeated but values for the `subject` column are now atomic for each record/row.

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

What is Second Normal Form?

For a table to be in the Second Normal Form, it must satisfy two conditions:

1. The table should be in the First Normal Form.
2. There should be no Partial Dependency.

If you want you can skip the video, as the concept is covered in detail below the video.

What is **Partial Dependency**? Do not worry about it. First let's understand what is **Dependency** in a table?

What is Dependency?

Let's take an example of a **Student** table with columns `student_id`, `name`, `reg_no`(registration number), `branch` and `address`(student's home address).

student_id	name	reg_no	branch	address

In this table, `student_id` is the primary key and will be unique for every row, hence we can use `student_id` to fetch any row of data from this table

Even for a case, where student names are same, if we know the `student_id` we can easily fetch the correct record.

student_id	name	reg_no	branch	address

10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat

Hence we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.

I can ask from branch name of student with `student_id` **10**, and I can get it. Similarly, if I ask for name of student with `student_id` **10** or **11**, I will get it. So all I need is `student_id` and every other column **depends** on it, or can be fetched using it.

This is **Dependency** and we also call it **Functional Dependency**.

What is Partial Dependency?

Now that we know what dependency is, we are in a better state to understand what partial dependency is.

For a simple table like Student, a single column like `student_id` can uniquely identfy all the records in a table.

But this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.

Let's create another table for **Subject**, which will have `subject_id` and `subject_name` fields and `subject_id` will be the primary key.

subject_id	subject_name
1	Java
2	C++
3	Php

Now we have a **Student** table with student information and another table **Subject** for storing subject information.

Let's create another table **Score**, to store the **marks** obtained by students in the respective subjects. We will also be saving **name of the teacher** who teaches that subject along with marks.

score_id	student_id	subject_id	marks	teacher
1	10	1	70	Java Teacher

2	10	2	75	C++ Teacher
3	11	1	80	Java Teacher

In the score table we are saving the **student_id** to know which student's marks are these and **subject_id** to know for which subject the marks are for.

Together, `student_id + subject_id` forms a **Candidate Key**(learn about Database Keys) for this table, which can be the **Primary key**.

Confused, How this combination can be a primary key?

See, if I ask you to get me marks of student with `student_id` 10, can you get it from this table? No, because you don't know for which subject. And if I give you `subject_id`, you would not know for which student. Hence we need `student_id + subject_id` to uniquely identify any row.

But where is Partial Dependency?

Now if you look at the **Score** table, we have a column names `teacher` which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is `student_id` & `subject_id` but the teacher's name only depends on subject, hence the `subject_id`, and has nothing to do with `student_id`.

This is **Partial Dependency**, where an attribute in a table depends on only a part of the primary key and not on the whole key.

How to remove Partial Dependency?

There can be many different solutions for this, but out objective is to remove teacher's name from Score table.

The simplest solution is to remove columns `teacher` from Score table and add it to the Subject table. Hence, the Subject table will become:

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

And our Score table is now in the second normal form, with no partial dependency.

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11	1	80

Quick Recap

1. For a table to be in the Second Normal form, it should be in the First Normal form and it should not have Partial Dependency.
2. Partial Dependency exists, when for a composite primary key, any attribute in the table depends only on a part of the primary key and not on the complete primary key.
3. To remove Partial dependency, we can divide the table, remove the attribute which is causing partial dependency, and move it to some other table where it fits in well.

Third Normal Form (3NF)

Third Normal Form is an upgrade to Second Normal Form. When a table is in the Second Normal Form and has no transitive dependency, then it is in the Third Normal Form.

Before moving forward with Third Normal Form, check these topics out to understand the concept better :

- First Normal Form (1NF)
- Second Normal Form (2NF)

The video below covers the concept of Third Normal Form in details.

In our last tutorial, we learned about the second normal form and even normalized our **Score** table into the 2nd Normal Form.

So let's use the same example, where we have 3 tables, **Student**, **Subject** and **Score**.

Student Table

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat
12	Bkon	09-WY	IT	Rajasthan

Subject Table

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

Score Table

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11	1	80

In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

score_id	student_id	subject_id	marks	exam_name	total_marks

Requirements for Third Normal Form

For a table to be in the third normal form,

1. It should be in the Second Normal form.
2. And it should not have Transitive Dependency.

What is Transitive Dependency?

With `exam_name` and `total_marks` added to our Score table, it saves more data now. Primary key for our Score table is a composite key, which means it's made up of two attributes or columns → **student_id + subject_id**.

Our new column `exam_name` depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Prctical exams and for some you don't. So we can say that `exam_name` is dependent on both `student_id` and `subject_id`.

And what about our second new column total_marks? Does it depend on our Score table's primary key?

Well, the column total_marks depends on exam_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total_marks depends on it.

This is **Transitive Dependency**. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

How to remove Transitive Dependency?

Again the solution is very simple. Take out the columns exam_name and total_marks from Score table and put them in an **Exam** table and use the exam_id wherever required.

Score Table: In 3rd Normal Form

score_id	student_id	subject_id	marks	exam_id

The new Exam table

exam_id	exam_name	total_marks
1	Workshop	200
2	Mains	70
3	Practicals	30

Advantage of removing Transitive Dependency

The advantage of removing transitive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form or BCNF is an extension to the third normal form, and is also known as 3.5 Normal Form.

Before you continue with Boyce-Codd Normal Form, check these topics for better understanding of database normalization concept:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)

Follow the video above for complete explanation of BCNF. Or, if you want, you can even skip the video and jump to the section below for the complete tutorial.

In our last tutorial, we learned about the third normal form and we also learned how to remove **transitive dependency** from a table, we suggest you to follow the last tutorial before this one.

Rules for BCNF

For a table to satisfy the Boyce-Codd Normal Form, it should satisfy the following two conditions:

1. It should be in the **Third Normal Form**.
2. And, for any dependency A → B, A should be a **super key**.

The second point sounds a bit tricky, right? In simple words, it means, that for a dependency A → B, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

Time for an Example

Below we have a college enrolment table with columns `student_id`, `subject` and `professor`.

student_id	subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

As you can see, we have also added some sample data to the table.

In the table above:

- One student can enrol for multiple subjects. For example, student with **student_id** 101, has opted for subjects - Java & C++
- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like we have for Java.

What do you think should be the **Primary Key**?

Well, in the table above `student_id, subject` together form the primary key, because using `student_id` and `subject`, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between `subject` and `professor` here, where `subject` depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as their is no **Partial Dependency**.

And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**.

Why this table is not in BCNF?

In the table above, `student_id, subject` form primary key, which means `subject` column is a **prime attribute**.

But, there is one more dependency, `professor` → `subject`.

And while `subject` is a prime attribute, `professor` is a **non-prime attribute**, which is not allowed by BCNF.

How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.

Below we have the structure for both the tables.

Student Table

student_id	p_id
101	1
101	2
and so on...	

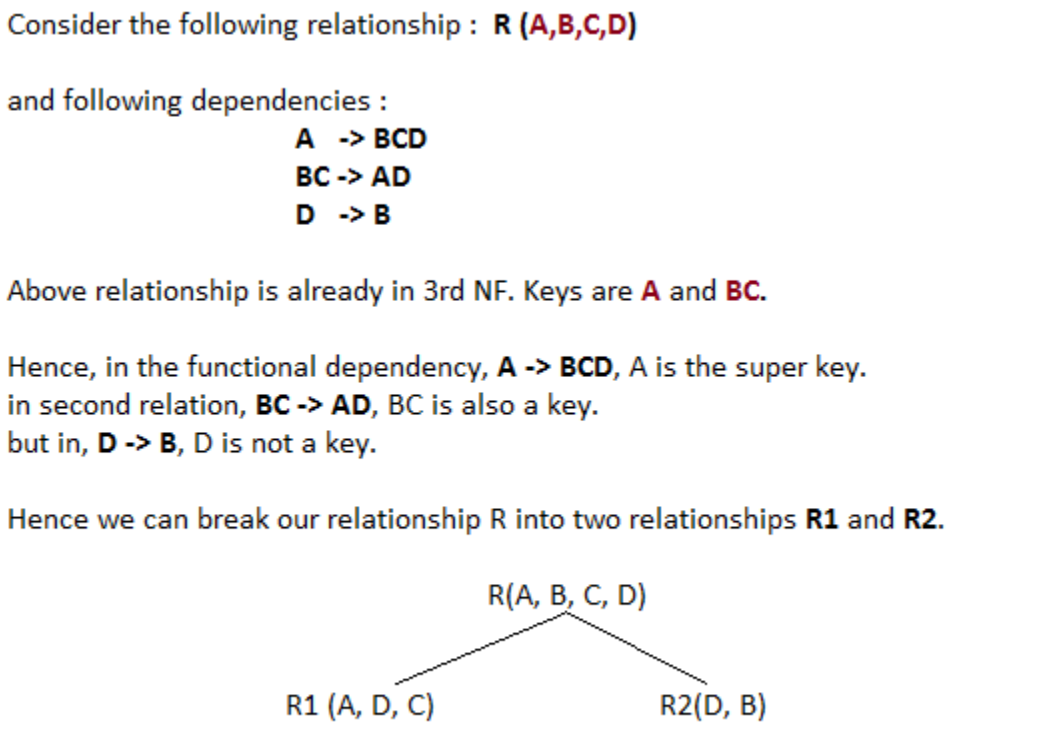
And, **Professor Table**

p_id	professor	subject
1	P.Java	Java
2	P.Cpp	C++
and so on...		

And now, this relation satisfy Boyce-Codd Normal Form. In the next tutorial we will learn about the **Fourth Normal Form**.

A more Generic Explanation

In the picture below, we have tried to explain BCNF in terms of relations.



Fourth Normal Form (4NF)

Fourth Normal Form comes into picture when **Multi-valued Dependency** occur in any relation. In this tutorial we will learn about Multi-valued Dependency, how to remove it and how to make any table satisfy the fourth normal form.

Follow the video above for complete explanation of 4th Normal Form. Or, if you want, you can even skip the video and jump to the section below for the complete tutorial.

In our last tutorial, we learned about the **boyce-codd normal form**, we suggest you to follow the last tutorial before this one.

Rules for 4th Normal Form

For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:

1. It should be in the **Boyce-Codd Normal Form**.
2. And, the table should not have any **Multi-valued Dependency**.

Let's try to understand what multi-valued dependency is in the next section.

What is Multi-valued Dependency?

A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation `R(A,B,C)`, if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

Time for an Example

Below we have a college enrolment table with columns `s_id`, `course` and `hobby`.

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey
2	C#	Cricket
2	Php	Hockey

As you can see in the table above, student with `s_id` **1** has opted for two courses, **Science** and **Maths**, and has two hobbies, **Cricket** and **Hockey**.

You must be thinking what problem this can lead to, right?

Well the two records for student with `s_id` **1**, will give rise to two more records, as shown below, because for one student, two hobbies exists, hence along with both the courses, these hobbies should be specified.

s_id	course	hobby
1	Science	Cricket

1	Maths	Hockey
1	Science	Hockey
1	Maths	Cricket

And, in the table above, there is no relationship between the columns `course` and `hobby`. They are independent of each other.

So there is multi-value dependency, which leads to un-necessary repetition of data and other anomalies as well.

How to satisfy 4th Normal Form?

To make the above relation satify the 4th normal form, we can decompose the table into 2 tables.

CourseOpted Table

s_id	course
1	Science
1	Maths
2	C#
2	Php

And, **Hobbies Table**,

s_id	hobby
1	Cricket
1	Hockey
2	Cricket

2	Hockey

Now this relation satisfies the fourth normal form.

A table can also have functional dependency along with multi-valued dependency. In that case, the functionally dependent columns are moved in a separate table and the multi-valued dependent columns are moved to separate tables.

If you design your database carefully, you can easily avoid these issues.

Fifth Normal Form (5NF)

Fifth Normal Form in Database Normalization is generally not implemented in real life database design. But you should know what it it.

Introduction to SQL

Structure Query Language(SQL) is a database query language used for storing and managing data in Relational DBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model of database. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) use **SQL** as the standard database query language. SQL is used to perform all types of data operations in RDBMS.

SQL Command

SQL defines following ways to manipulate data stored in an RDBMS.

DDL: Data Definition Language
This includes changes to the structure of the table like creation of table, altering table, deleting a table etc.

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

Command	Description
create	to create new table or database
alter	for alteration
truncate	delete data from table
drop	to drop a table

rename	to rename a table

DML: Data Manipulation Language

DML commands are used for manipulating the data stored in the table and not the table itself.

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

Command	Description
insert	to insert a new row
update	to update existing row
delete	to delete a row
merge	merging two rows or two tables

TCL: Transaction Control Language

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling the data back to its original state. It can also make any temporary change permanent.

Command	Description
commit	to permanently save
rollback	to undo change
savepoint	to save temporarily

DCL: Data Control Language

Data control language are the commands to grant and take back authority from any database user.

Command	Description
grant	grant permission of right
revoke	take back permission.

DQL: Data Query Language

Data query language is used to fetch data from tables based on conditions that we can easily apply.

Command	Description
select	retrieve records from one or more table

SQL: create command

create is a DDL SQL command used to create a table or a database in relational database management system.

Creating a Database

To create a database in RDBMS, **create** command is used. Following is the syntax,

```
CREATE DATABASE <DB_NAME>;
```

Copy

Example for creating Database

```
CREATE DATABASE Test;
```

Copy

The above command will create a database named **Test**, which will be an empty schema without any table.

To create tables in this newly created database, we can again use the create command.

Creating a Table

`create` command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the **names** and **datatypes** of various columns in the `create` command itself.

Following is the syntax,

```
CREATE TABLE <TABLE_NAME>
(
    column_name1 datatype1,
    column_name2 datatype2,
    column_name3 datatype3,
    column_name4 datatype4
);
```

Copy

`create` table command will tell the database system to create a new table with the given table name and column information.

Example for creating Table

```
CREATE TABLE Student(
    student_id INT,
    name VARCHAR(100),
    age INT);
```

Copy

The above command will create a new table with name **Student** in the current database with 3 columns, namely `student_id`, `name` and `age`. Where the column `student_id` will only store integer, `name` will hold upto 100 characters and `age` will again store only integer value.

If you are currently not logged into your database in which you want to create the table then you can also add the database name along with table name, using a dot operator `.`

For example, if we have a database with name **Test** and we want to create a table **Student** in it, then we can do so using the following query:

```
CREATE TABLE Test.Student(
    student_id INT,
    name VARCHAR(100),
    age INT);
```

Copy

Most commonly used datatypes for Table columns

Here we have listed some of the most commonly used datatypes used for columns in tables.

Datatype	Use
INT	used for columns which will store integer values.
FLOAT	used for columns which will store float values.
DOUBLE	used for columns which will store float values.
VARCHAR	used for columns which will be used to store characters and integers, basically a string.
CHAR	used for columns which will store char values(single character).
DATE	used for columns which will store date values.
TEXT	used for columns which will store text which is generally long in length. For example, if you create a table for storing profile information of a social networking website, then for **about me** section you can have a column of type `TEXT`.

SQL: `ALTER` command

`alter` command is used for altering the table structure, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.
- to drop a column from the table.

`ALTER` Command: Add a new Column

Using `ALTER` command we can add a column to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD(
```

```
    column_name datatype);
```

Copy

Here is an Example for this,

```
ALTER TABLE student ADD(

    address VARCHAR(200)

);
```

Copy

The above command will add a new column address to the table **student**, which will hold data of type varchar which is nothing but string, of length 200.

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

```
ALTER TABLE table_name ADD(

    column_name1 datatype1,

    column-name2 datatype2,

    column-name3 datatype3);
```

Copy

Here is an Example for this,

```
ALTER TABLE student ADD(

    father_name VARCHAR(60),

    mother_name VARCHAR(60),

    dob DATE);
```

Copy

The above command will add three new columns to the **student** table

---

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

```
ALTER TABLE table_name ADD(

    column-name1 datatype1 DEFAULT some_value

);
```

Copy

Here is an Example for this,

```
ALTER TABLE student ADD(

    dob DATE DEFAULT '01-Jan-99'

);
```

Copy

The above command will add a new column with a preset default value to the table **student**.

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

```
ALTER TABLE table_name modify(

    column_name datatype

);
```

Copy

Here is an Example for this,

```
ALTER TABLE student MODIFY(

    address varchar(300));
```

Copy

Remember we added a new column address in the beginning? The above command will modify the address column of the **student** table, to now hold upto 300 characters.

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

```
ALTER TABLE table_name RENAME

    old_column_name TO new_column_name;
```

Copy

Here is an example for this,

```
ALTER TABLE student RENAME

    address TO location;
```

Copy

The above command will rename address column to location.

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns. Following is the syntax,

```
ALTER TABLE table_name DROP(

    column_name);
```

Copy

Here is an example for this,

```
ALTER TABLE student DROP(

    address);
```

Copy

The above command will drop the address column from the table **student**.

SQL Truncate, Drop or Rename a Table

In this tutorial we will learn about the various DDL commands which are used to re-define the tables.

TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized. Following is its syntax,

```
TRUNCATE TABLE table_name
```

Copy

Here is an example explaining it,

```
TRUNCATE TABLE student;
```

Copy

The above query will delete all the records from the table **student**.

In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command. We will also learn about the difference between the two in that tutorial.

DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it. Following is its syntax,

```
DROP TABLE table_name
```

Copy

Here is an example explaining it,

```
DROP TABLE student;
```

Copy

The above query will delete the **Student** table completely. It can also be used on Databases, to delete the complete database. For example, to drop a database,

```
DROP DATABASE Test;
```

Copy

The above query will drop the database with name **Test** from the system.

RENAME query

RENAME command is used to set a new name for any existing table. Following is the syntax,

```
RENAME TABLE old_table_name to new_table_name
```

Copy

Here is an example explaining it.

```
RENAME TABLE student to students_info;
```

Copy

The above query will rename the table **student** to **students_info**.

Using INSERT SQL command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

Talking about the Insert command, whenever we post a Tweet on Twitter, the text is stored in some table, and as we post a new tweet, a new record gets inserted in that table.

INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

```
INSERT INTO table_name VALUES(data1, data2, ...)
```

Copy

Lets see an example,

Consider a table **student** with the following fields.

s_id	name	age

```
INSERT INTO student VALUES(101, 'Adam', 15);
```

Copy

The above command will insert a new record into **student** table.

s_id	name	age
101	Adam	15

Insert value into only specific columns

We can use the `INSERT` command to insert values for only some specific columns of a row. We can specify the column names along with the values to be inserted like this,

```
INSERT INTO student(id, name) values(102, 'Alex');
```

Copy

The above SQL query will only insert id and name values in the newly inserted record.

Insert NULL value to a column

Both the statements below will insert `NULL` value into **age** column of the **student** table.

```
INSERT INTO student(id, name) values(102, 'Alex');
```

Copy

Or,

```
INSERT INTO Student VALUES(102,'Alex', null);
```

Copy

The above command will insert only two column values and the other column is set to null.

S_id	S_Name	age
101	Adam	15

102	Alex	

Insert Default value to a column

```
INSERT INTO Student VALUES(103,'Chris', default)
```

Copy

S_id	S_Name	age
101	Adam	15
102	Alex	
103	chris	14

Suppose the column `age` in our tabel has a default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

```
INSERT INTO Student VALUES(103,'Chris')
```

Copy

Using `UPDATE` SQL command

Let's take an example of a real-world problem. These days, Facebook provides an option for **Editing** your status update, how do you think it works? Yes, using the **Update** SQL command.

Let's learn about the syntax and usage of the `UPDATE` command.

`UPDATE` command

`UPDATE` command is used to update any record of data in a table. Following is its general syntax,

```
UPDATE table_name SET column_name = new_value WHERE some_condition;
```

Copy

`WHERE` is used to add a condition to any SQL query, we will soon study about it in detail.

Lets take a sample table **student**,

student_id	name	age
101	Adam	15
102	Alex	
103	chris	14

```
UPDATE student SET age=18 WHERE student_id=102;
```

Copy

S_id	S_Name	age
101	Adam	15
102	Alex	18
103	chris	14

In the above statement, if we do not use the `WHERE` clause, then our update query will update age for all the columns of the table to **18**.

Updating Multiple Columns

We can also update values of multiple columns using a single `UPDATE` statement.

```
UPDATE student SET name='Abhi', age=17 where s_id=103;
```

Copy

The above command will update two columns of the record which has `s_id` 103.

s_id	name	age
101	Adam	15
102	Alex	18

103	Abhi	17

`UPDATE` Command: Incrementing Integer Value

When we have to update any integer value in a table, then we can fetch and update the value in the table in a single statement.

For example, if we have to update the `age` column of **student** table every year for every student, then we can simply run the following `UPDATE` statement to perform the following operation:

```
UPDATE student SET age = age+1;
```

Copy

As you can see, we have used `age = age + 1` to increment the value of age by 1.

NOTE: This style only works for integer values.

Using `DELETE` SQL command

When you ask any question in [Studytonight's Forum] it gets saved into a table. And using the **Delete** option, you can even delete a question asked by you. How do you think that works? Yes, using the Delete DML command.

Let's study about the syntax and the usage of the Delete command.

`DELETE` command

`DELETE` command is used to delete data from a table.

Following is its general syntax,

```
DELETE FROM table_name;
```

Copy

Let's take a sample table **student**:

s_id	name	age
101	Adam	15
102	Alex	18
103	Abhi	17

Delete all Records from a Table

```
DELETE FROM student;
```

Copy

The above command will delete all the records from the table **student**.

Delete a particular Record from a Table

In our **student** table if we want to delete a single record, we can use the `WHERE` clause to provide a condition in our `DELETE` statement.

```
DELETE FROM student WHERE s_id=103;
```

Copy

The above command will delete the record where `s_id` is 103 from the table **student**.

S_id	S_Name	age
101	Adam	15
102	Alex	18

Isn't `DELETE` same as `TRUNCATE`

`TRUNCATE` command is different from `DELETE` command. The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).

For eg: If you have a table with 10 rows and an **auto_increment** primary key, and if you use `DELETE` command to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the `DELETE` command, the auto_increment primary key will start from 11. But in case of `TRUNCATE` command, primary key is re-initialized, and it will again start from 1.

Commit, Rollback and Savepoint SQL commands

Transaction Control Language(TCL) commands are used to manage transactions in the database.

Before moving forward with TCL commands, check these topics out first:

- DML commands
- DDL COMMAND

These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

COMMIT command

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's syntax,

```
COMMIT;
```

Copy

ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

Following is rollback command's syntax,

```
ROLLBACK TO savepoint_name;
```

Copy

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

```
SAVEPOINT savepoint_name;
```

Copy

In short, using this command we can **name** the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

Using Savepoint and Rollback

Following is the table **class**,

id	name
1	Abhi
2	Adam
4	Alex

Lets use some SQL queries on the above table and see the results.

```
INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C;

SELECT * FROM class;
```

Copy

NOTE: `SELECT` statement is used to show the data stored in the table.

The resultant table will look like,

id	name

1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

```
ROLLBACK TO B;


SELECT * FROM class;
```

Copy

Now our **class** table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

```
ROLLBACK TO A;


SELECT * FROM class;
```

Copy

Now the table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

So now you know how the commands `COMMIT`, `ROLLBACK` and `SAVEPOINT` works.

Using `GRANT` and `REVOKE`

Data Control Language(DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges. Privileges are of two types,

- **System:** This includes permissions for creating session, table, etc and all types of other system privileges.
- **Object:** This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

- `GRANT`: Used to provide any user access privileges or other priviliges for the database.
- `REVOKE`: Used to take back permissions from any user.

Allow a User to create session

When we create a user in SQL, it is not even allowed to login and create a session until and unless proper permissions/priviliges are granted to the user.

Following command can be used to grant the session creating priviliges.

```
GRANT CREATE SESSION TO username;
```

Copy

Allow a User to create table

To allow a user to create tables in the database, we can use the below command,

```
GRANT CREATE TABLE TO username;
```

Copy

Provide user with space on tablespace to store table

Allowing a user to create table is not enough to start storing data in that table. We also must provide the user with priviliges to use the available tablespace for their table and data.

```
ALTER USER username QUOTA UNLIMITED ON SYSTEM;
```

Copy

The above command will alter the user details and will provide it access to unlimited tablespace on system.

NOTE: Generally unlimited quota is provided to Admin users.

Grant all privilege to a User

`sysdba` is a set of priviliges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the `sysdba` permission.

```
GRANT sysdba TO username
```

Copy

Grant permission to create any table

Sometimes user is restricted from creating come tables with names which are reserved for system tables. But we can grant privileges to a user to create any table using the below command,

```
GRANT CREATE ANY TABLE TO username
```

Copy

Grant permission to drop any table

As the title suggests, if you want to allow user to drop any table from the database, then grant this privilege to the user,

```
GRANT DROP ANY TABLE TO username
```

Copy

To take back Permissions

And, if you want to take back the privileges from any user, use the `REVOKE` command.

```
REVOKE CREATE TABLE FROM username
```

Copy

Using the `WHERE` SQL clause

`WHERE` clause is used to specify/apply any condition while retrieving, updating or deleting data from a table. This clause is used mostly with `SELECT`, `UPDATE` and `DELETE`query.

When we specify a condition using the `WHERE` clause then the query executes only for those records for which the condition specified by the `WHERE` clause is true.

Syntax for `WHERE` clause

Here is how you can use the `WHERE` clause with a `DELETE` statement, or any other statement,

```
DELETE FROM table_name WHERE [condition];
```

Copy

The `WHERE` clause is used at the end of any SQL query, to specify a condition for execution.

Time for an Example

Consider a table **student**,

s_id	name	age	address
101	Adam	15	Chennai
102	Alex	18	Delhi
103	Abhi	17	Banglore
104	Ankit	22	Mumbai

Now we will use the `SELECT` statement to display data of the table, based on a condition, which we will add to our `SELECT` query using `WHERE` clause.

Let's write a simple SQL query to display the record for student with `s_id` as 101.

```
SELECT s_id,
```

```
    name,

    age,

    address

    FROM student WHERE s_id = 101;
```

Copy

Following will be the result of the above query.

s_id	name	age	address
101	Adam	15	Noida

Applying condition on Text Fields

In the above example we have applied a condition to an integer value field, but what if we want to apply the condition on `name` field. In that case we must enclose the value in single quote `''`. Some databases even accept double quotes, but single quotes is accepted by all.

```
SELECT s_id,

    name,

    age,

    address

    FROM student WHERE name = 'Adam';
```

Copy

Following will be the result of the above query.

s_id	name	age	address
101	Adam	15	Noida

Operators for `WHERE` clause condition

Following is a list of operators that can be used while specifying the `WHERE` clause condition.

Operator	Description

=	Equal to
!=	Not Equal to
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greate than or Equal to
BETWEEN	Between a specified range of values
LIKE	This is used to search for a pattern in value.
IN	In a given set of values

SQL LIKE clause

LIKE clause is used in the condition in SQL query with the WHERE clause. LIKE clause compares data with an expression using wildcard operators to match pattern given in the condition.

Wildcard operators

There are two wildcard operators that are used in LIKE clause.

- **Percent sign %**: represents zero, one or more than one character.
- **Underscore sign _**: represents only a single character.

Example of LIKE clause

Consider the following **Student** table.

s_id	s_Name	age

101	Adam	15
102	Alex	18
103	Abhi	17

```
SELECT * FROM Student WHERE s_name LIKE 'A%';
```

Copy

The above query will return all records where **s_name** starts with character 'A'.

s_id	s_Name	age
101	Adam	15
102	Alex	18
103	Abhi	17

Using _ and %

```
SELECT * FROM Student WHERE s_name LIKE '_d%';
```

Copy

The above query will return all records from **Student** table where **s_name** contain 'd' as second character.

s_id	s_Name	age
101	Adam	15

Using % only

```
SELECT * FROM Student WHERE s_name LIKE '%x';
```

Copy

The above query will return all records from **Student** table where **s_name** contain 'x' as last character.

s_id	s_Name	age
102	Alex	18

SQL ORDER BY Clause

Order by clause is used with SELECT statement for arranging retrieved data in sorted order.
The **Order by** clause by default sorts the retrieved data in ascending order. To sort the data in descending order DESC keyword is used with Order by clause.

Syntax of Order By

```
SELECT column-list|* FROM table-name ORDER BY ASC | DESC;
```

Copy

Using default Order by

Consider the following **Emp** table,

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

```
SELECT * FROM Emp ORDER BY salary;
```

Copy

The above query will return the resultant data in ascending order of the **salary**.

eid	name	age	salary
403	Rohan	34	6000
402	Shane	29	8000
405	Tiger	35	8000
401	Anu	22	9000
404	Scott	44	10000

Using Order by DESC

Consider the **Emp** table described above,

```
SELECT * FROM Emp ORDER BY salary DESC;
```

Copy

The above query will return the resultant data in descending order of the **salary**.

eid	name	age	salary
404	Scott	44	10000
401	Anu	22	9000
405	Tiger	35	8000
402	Shane	29	8000
403	Rohan	34	6000

Check out other DCL commands and their usage:

- SELECT query
- WHERE clause
- LIKE clause
- Group BY clause

- Having clause

SQL Group By Clause

Group by clause is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Syntax for using Group by in a statement.

```
SELECT column_name, function(column_name)

FROM table_name

WHERE condition

GROUP BY column_name
```

Copy

Example of Group by in a Statement

Consider the following **Emp** table.

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000
405	Tiger	35	8000

Here we want to find **name** and **age** of employees grouped by their **salaries** or in other words, we will be grouping employees based on their salaries, hence, as a result, we will get a data set, with unique salaries listed, along side the first employee's name and age to have that salary. Hope you are getting the point here!

group by is used to group different row of data together based on any one column.

SQL query for the above requirement will be,

```
SELECT name, age
```

```
FROM Emp GROUP BY salary
```

Copy

Result will be,

name	age
Rohan	34
Shane	29
Anu	22

Example of `Group by` in a Statement with `WHERE` clause

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000
405	Tiger	35	8000

SQL query will be,

```
SELECT name, salary

FROM Emp

WHERE age > 25

GROUP BY salary
```

Copy

Result will be.

name	salary
Rohan	6000
Shane	8000
Scott	9000

You must remember that `Group By` clause will always come at the end of the SQL query, just like the `Order by` clause.

SQL `HAVING` Clause

Having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in `Group by` based SQL queries, just like `WHERE` clause is used with `SELECT` query.

Syntax for `HAVING` clause is,

```
SELECT column_name, function(column_name)

FROM table_name

WHERE column_name condition

GROUP BY column_name

HAVING function(column_name) condition
```

Copy

Example of SQL Statement using `HAVING`

Consider the following **Sale** table.

oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi

14	ord4	1000	Adam
15	ord5	2000	Alex

Suppose we want to find the **customer** whose **previous_balance** sum is more than **3000**.

We will use the below SQL query,

```
SELECT *

FROM sale GROUP BY customer

HAVING sum(previous_balance) > 3000
```

Copy

Result will be,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex

The main objective of the above SQL query was to find out the name of the customer who has had a **previous_balance** more than **3000**, based on all the previous sales made to the customer, hence we get the first row in the table for customer Alex.

DISTINCT keyword

The distinct keyword is used with SELECT statement to retrieve unique values from the table. Distinct removes all the duplicate records while retrieving records from any table in the database.

Syntax for DISTINCT Keyword

```
SELECT DISTINCT column-name FROM table-name;
```

Copy

Example using DISTINCT Keyword

Consider the following **Emp** table. As you can see in the table below, there is employee **name**, along with employee **salary** and **age**.

In the table below, multiple employees have the same salary, so we will be using DISTINCT keyword to list down distinct salary amount, that is currently being paid to the employees.

eid	name	age	salary
401	Anu	22	5000
402	Shane	29	8000
403	Rohan	34	10000
404	Scott	44	10000
405	Tiger	35	8000

```
SELECT DISTINCT salary FROM Emp;
```

Copy

The above query will return only the unique salary from **Emp** table.

salary
5000
8000
10000

SQL AND & OR operator

The AND and OR operators are used with the WHERE clause to make more precise conditions for fetching data from database by combining more than one condition together.

AND operator

AND operator is used to set multiple conditions with the WHERE clause, alongside, SELECT, UPDATE or DELETE SQL queries.

Example of AND operator

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	5000
402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

```
SELECT * FROM Emp WHERE salary < 10000 AND age > 25
```

The above query will return records where **salary** is less than **10000** and **age** greater than **25**. Hope you get the concept here. We have used the AND operator to specify two conditions with WHERE clause.

eid	name	age	salary
402	Shane	29	8000
405	Tiger	35	9000

OR operator

OR operator is also used to combine multiple conditions with WHERE clause. The only difference between AND and OR is their behaviour.

When we use AND to combine two or more than two conditions, records satisfying all the specified conditions will be there in the result.

But in case of OR operator, atleast one condition from the conditions specified must be satisfied by any record to be in the resultset.

Example of OR operator

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	5000
402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

```
SELECT * FROM Emp WHERE salary > 10000 OR age > 25
```

The above query will return records where **either** salary is greater than 10000 **or** age is greater than 25.

402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

Division Operator in SQL

The division operator is used when we have to evaluate queries which contain the keyword ALL.

Some instances where division operator is used are:

1. Which person has account in all the banks of a particular city?
2. Which students have taken all the courses required to graduate?

In above specified problem statements, the description after the keyword 'all' defines a set which contains some elements and the final result contains those units which satisfy these requirements.

Another way how you can identify the usage of division operator is by using the logical implication of `if...then`. In context of the above two examples, we can see that the queries mean that,

1. If there is a bank in that particular city, that person must have an account in that bank.
2. If there is a course in the list of courses required to be graduated, that person must have taken that course.

Do not worry if you are not clear with all this new things right away, we will try to expain as we move on with this tutorial.

We shall see the second example, mentioned above, in detail.

Table 1: Course_Taken → It consists of the names of Students against the courses that they have taken.

Student_Name	Course
Robert	Databases
Robert	Programming Languages
David	Databases
David	Operating Systems
Hannah	Programming Languages
Hannah	Machine Learning
Tom	Operating Systems

Table 2: Course_Required → It consists of the courses that one is required to take in order to graduate.

Course
Databases
Programming Languages

Using Division Operator

So now, let's try to find out the correct SQL query for getting results for the first requirement, which is:

Query: Find all the students who can graduate. (i.e. who have taken all the subjects required for one to graduate.)

Unfortunately, there is no direct way by which we can express the division operator. Let's walk through the steps, to write the query for the division operator.

1. Find all the students

Create a set of all students that have taken courses. This can be done easily using the following command.

```
CREATE TABLE AllStudents AS SELECT DISTINCT Student_Name FROM
Course_Taken
```

Copy

This command will return the table **AllStudents**, as the resultset:

Student_name
Robert
David
Hannah
Tom

2. Find all the students and the courses required to graduate

Next, we will create a set of students and the courses they need to graduate. We can express this in the form of Cartesian Product of **AllStudents** and **Course_Required** using the following command.

```
CREATE table StudentsAndRequired AS

SELECT AllStudents.Student_Name, Course_Required.Course

FROM AllStudents, Course_Required
```

Copy

Now the new resultset - table **StudentsAndRequired** will be:

Student_Name	Course
Robert	Databases
Robert	Programming Languages
David	Databases
David	Programming Languages
Hannah	Databases
Hannah	Programming Languages
Tom	Databases
Tom	Programming Languages

---

3. Find all the students and the required courses they have not taken

Here, we are taking our first step for finding the students who cannot graduate. The idea is to simply find the students who have not taken certain courses that are required for graduation and hence they wont be able to graduate. This is simply all those tuples/rows which are present in **StudentsAndRequired** and not present in **Course_Taken**.

```
CREATE  table StudentsAndNotTaken AS

SELECT * FROM StudentsAndRequired WHERE NOT EXISTS

(Select * FROM Course_Taken WHERE StudentsAndRequired.Student_Name =
Course_Taken.Student_Name

AND StudentsAndRequired.Course = Course_Taken.Course)
```

Copy

The table **StudentsAndNotTaken** comes out to be:

Student_Name	Course

David	Programming Languages
Hannah	Databases
Tom	Databases
Tom	Programming Languages

---

4. Find all students who cannot graduate

All the students who are present in the table **StudentsAndNotTaken** are the ones who cannot graduate. Therefore, we can find the students who cannot graduate as,

```
CREATE table CannotGraduate AS SELECT DISTINCT Student_Name FROM
StudentsAndNotTaken
```

Copy

Student_name
David
Hannah
Tom

---

5. Find all students who can graduate

The students who can graduate are simply those who are present in **AllStudents** but not in **CannotGraduate**. This can be done by the following query:

```
CREATE Table CanGraduate AS SELECT * FROM AllStudents

WHERE NOT EXISTS

(SELECT * FROM CannotGraduate WHERE

    CannotGraduate.Student_name = AllStudents.Student_name)
```

Copy

The results will be as follows:

Student_name
Robert

Hence we just learned, how different steps can lead us to the final answer. Now let us see how to write all these 5 steps in one single query so that we do not have to create so many tables.

```
SELECT DISTINCT  x.Student_Name FROM Course_Taken AS x WHERE NOT

EXISTS(SELECT * FROM Course_Required AS y WHERE NOT

EXISTS(SELECT * FROM Course_Taken AS z

    WHERE z.Student_name = x.Student_name

    AND z.Course = y.Course ))
```

Copy

Student_name
Robert

This gives us the same result just like the 5 steps above.

SQL Constraints

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into the following two types,

1. **Column level constraints:** Limits only column data.
2. **Table level constraints:** Limits whole table data.

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, use the NOT NULL constraint.

- It restricts a column from having a NULL value.
- We use ALTER statement and MODIFY statement to specify this constraint.

One important point to note about this constraint is that it cannot be defined at table level.

Example using NOT NULL constraint:

```
CREATE TABLE Student

(     s_id int NOT NULL,

      name varchar(60),

      age  int

);
```

Copy

The above query will declare that the **s_id** field of **Student** table will not take NULL value.

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

```
ALTER TABLE Student

MODIFY s_id int NOT NULL;
```

Copy

UNIQUE Constraint

It ensures that a column will only have unique values. A UNIQUE constraint field cannot have any duplicate data.

- It prevents two records from having identical values in a column
- We use ALTER statement and MODIFY statement to specify this constraint.

Example of UNIQUE Constraint:

Here we have a simple CREATE query to create a table, which will have a column **s_id** with unique values.

```
CREATE TABLE Student

(     s_id int  NOT NULL,

      name varchar(60),

      age int  NOT NULL UNIQUE

);
```

Copy

The above query will declare that the **s_id** field of **Student** table will only have unique values and wont take NULL value.

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

```
ALTER TABLE Student

MODIFY age INT NOT NULL UNIQUE;
```

Copy

The above query specifies that **s_id** field of **Student** table will only have unique value.

Primary Key Constraint

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

PRIMARY KEY constraint at Table Level

```
CREATE table Student

(     s_id int PRIMARY KEY,

      Name varchar(60) NOT NULL,

      Age int);
```

Copy

The above command will creates a PRIMARY KEY on the `s_id`.

PRIMARY KEY constraint at Column Level

```
ALTER table Student

ADD PRIMARY KEY (s_id);
```

Copy

The above command will creates a PRIMARY KEY on the `s_id`.

Foreign Key Constraint

Foreign Key is used to relate two tables. The relationship between the two tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

- This is also called a referencing key.
- We use ALTER statement and ADD statement to specify this constraint.

To understand FOREIGN KEY, let's see its use, with help of the below tables:

Customer_Detail Table

c_id	Customer_Name	address
101	Adam	Noida
102	Alex	Delhi
103	Stuart	Rohtak

Order_Detail Table

Order_id	Order_Name	c_id
10	Order1	101
11	Order2	103
12	Order3	102

In **Customer_Detail** table, **c_id** is the primary key which is set as foreign key in **Order_Detail** table. The value that is entered in **c_id** which is set as foreign key in **Order_Detail** table must be present in **Customer_Detail** table where it is set as primary key. This prevents invalid data to be inserted into **c_id** column of **Order_Detail** table.

If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

FOREIGN KEY constraint at Table Level

```
CREATE table Order_Detail(

    order_id int PRIMARY KEY,

    order_name varchar(60) NOT NULL,

    c_id int FOREIGN KEY REFERENCES Customer_Detail(c_id)

);
```

Copy

In this query, **c_id** in table Order_Detail is made as foriegn key, which is a reference of **c_id** column in Customer_Detail table.

FOREIGN KEY constraint at Column Level

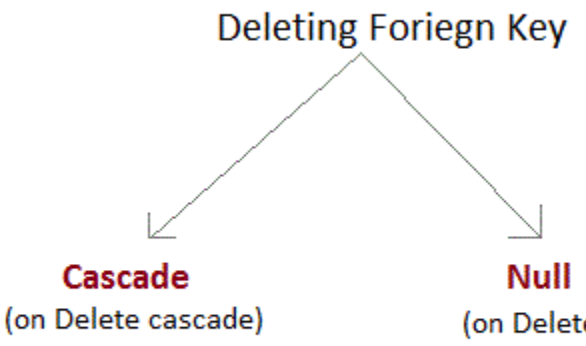
```
ALTER table Order_Detail

ADD FOREIGN KEY (c_id) REFERENCES Customer_Detail(c_id);
```

Copy

Behaviour of Foriegn Key Column on Delete

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in the main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which a record exits in the child table, then we must have some mechanism to save the integrity of data in the child table.



1. **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
2. **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is deleted from the main table.
3. If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

```
ERROR : Record in child table exist
```

CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

Using CHECK constraint at Table Level

```
CREATE table Student(

    s_id int NOT NULL CHECK(s_id > 0),

    Name varchar(60) NOT NULL,

    Age int

);
```

Copy

The above query will restrict the **s_id** value to be greater than zero.

Using `CHECK` constraint at Column Level

```
ALTER table Student ADD CHECK(s_id > 0);
```

Copy

Related Tutorials:

- SQL function
- SQL Join
- SQL Alias
- SQL SET operation
- SQL Sequences
- SQL Views

What are SQL Functions?

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two categories,

1. Aggregate Functions
2. Scalar Functions

Aggregate Functions

These functions **return a single value** after performing calculations on a group of values. Following are some of the frequently used Aggregrate functions.

`AVG()` **Function**

Average returns average value after calculating it from values in a numeric column.

Its general **syntax** is,

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

Copy

Using AVG() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000

402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find average salary will be,

```
SELECT avg(salary) from Emp;
```

Copy

Result of the above query will be,

avg(salary)
8200

COUNT() Function

Count returns the number of rows present in the table either based on some condition or without condition.

Its general **syntax** is,

```
SELECT COUNT(column_name) FROM table-name
```

Copy

Using COUNT() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000

403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to count employees, satisfying specified condition is,

```
SELECT COUNT(name) FROM Emp WHERE salary = 8000;
```

Copy

Result of the above query will be,

count(name)
2

Example of COUNT(distinct)

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query is,

```
SELECT COUNT(DISTINCT salary) FROM emp;
```

Copy

Result of the above query will be,

count(distinct salary)
4

FIRST() Function

First function returns first value of a selected column

Syntax for FIRST function is,

```
SELECT FIRST(column_name) FROM table-name;
```

Copy

Using FIRST() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query will be,

```
SELECT FIRST(salary) FROM Emp;
```

Copy

and the result will be,

first(salary)
9000

LAST() Function

LAST function returns the return last value of the selected column.

Syntax of LAST function is,

```
SELECT LAST(column_name) FROM table-name;
```

Copy

Using LAST() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query will be,

```
SELECT LAST(salary) FROM emp;
```

Copy

Result of the above query will be,

last(salary)
8000

MAX() Function

MAX function returns maximum value from selected column of the table.

Syntax of MAX function is,

```
SELECT MAX(column_name) from table-name;
```

Copy

Using MAX() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find the Maximum salary will be,

```
SELECT MAX(salary) FROM emp;
```

Copy

Result of the above query will be,

MAX(salary)
10000

MIN() Function

MIN function returns minimum value from a selected column of the table.

Syntax for MIN function is,

```
SELECT MIN(column_name) from table-name;
```

Copy

Using MIN() function

Consider the following **Emp** table,

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find minimum salary is,

```
SELECT MIN(salary) FROM emp;
```

Copy

Result will be,

MIN(salary)
6000

SUM() Function

SUM function returns total sum of a selected columns numeric values.

Syntax for SUM is,

```
SELECT SUM(column_name) from table-name;
```

Copy

Using SUM() function

Consider the following **Emp** table

eid	name	age	salary

401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SQL query to find sum of salaries will be,

```
SELECT SUM(salary) FROM emp;
```

Copy

Result of above query is,

SUM(salary)
41000

Scalar Functions

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions in SQL.

UCASE() Function

UCASE function is used to convert value of string column to Uppercase characters.

Syntax of UCASE,

```
SELECT UCASE(column_name) from table-name;
```

Copy

Using UCASE() function

Consider the following **Emp** table

eid	name	age	salary

401	anu	22	9000
402	shane	29	8000
403	rohan	34	6000
404	scott	44	10000
405	Tiger	35	8000

SQL query for using UCASE is,

```
SELECT UCASE(name) FROM emp;
```

Copy

Result is,

UCASE(name)
ANU
SHANE
ROHAN
SCOTT
TIGER

LCASE() Function

LCASE function is used to convert value of string columns to Lowecase characters.

Syntax for LCASE is,

```
SELECT LCASE(column_name) FROM table-name;
```

Copy

Using LCASE() function

Consider the following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	SCOTT	44	10000
405	Tiger	35	8000

SQL query for converting string value to Lower case is,

```
SELECT LCASE(name) FROM emp;
```

Copy

Result will be,

LCASE(name)
anu
shane
rohan
scott
tiger

MID() Function

MID function is used to extract substrings from column values of string type in a table.

Syntax for MID function is,

```
SELECT MID(column_name, start, length) from table-name;
```

Copy

Using MID() function

Consider the following **Emp** table

eid	name	age	salary
401	anu	22	9000
402	shane	29	8000
403	rohan	34	6000
404	scott	44	10000
405	Tiger	35	8000

SQL query will be,

```
SELECT MID(name,2,2) FROM emp;
```

Copy

Result will come out to be,

MID(name,2,2)
nu
ha
oh
co
ig

`ROUND()` Function

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values.

Syntax of Round function is,

```
SELECT ROUND(column_name, decimals) from table-name;
```

Copy

Using ROUND() function

Consider the following **Emp** table

eid	name	age	salary
401	anu	22	9000.67
402	shane	29	8000.98
403	rohan	34	6000.45
404	scott	44	10000
405	Tiger	35	8000.01

SQL query is,

```
SELECT ROUND(salary) from emp;
```

Copy

Result will be,

ROUND(salary)
9001
8001
6000

10000
8000

SQL JOIN

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It is used for combining column from two or more tables by using values common to both tables.

JOIN Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself, which is known as, **Self Join**.

Types of JOIN

Following are the types of JOIN that we can use in SQL:

- Inner
- Outer
- Left
- Right

Cross JOIN or Cartesian Product

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

```
SELECT column-name-list

FROM

table-name1 CROSS JOIN table-name2;
```

Copy

Example of Cross JOIN

Following is the **class** table,

ID	NAME
1	abhi
2	adam
4	alex

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Cross JOIN query will be,

```
SELECT * FROM

class CROSS JOIN class_info;
```

Copy

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	1	DELHI
4	alex	1	DELHI
1	abhi	2	MUMBAI
2	adam	2	MUMBAI

4	alex	2	MUMBAI
1	abhi	3	CHENNAI
2	adam	3	CHENNAI
4	alex	3	CHENNAI

As you can see, this join returns the cross product of all the records present in both the tables.

INNER Join or EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query.

Inner Join Syntax is,

```
SELECT column-name-list FROM

table-name1 INNER JOIN table-name2

WHERE table-name1.column-name = table-name2.column-name;
```

Copy

Example of INNER JOIN

Consider a **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Inner JOIN query will be,

```
SELECT * from class INNER JOIN class_info where class.id =
class_info.id;
```

Copy

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI

Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

The syntax for Natural Join is,

```
SELECT * FROM

table-name1 NATURAL JOIN table-name2;
```

Copy

Example of Natural JOIN

Here is the **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Natural join query will be,

```
SELECT * from class NATURAL JOIN class_info;
```

Copy

The resultset table will look like,

ID	NAME	Address
1	abhi	DELHI
2	adam	MUMBAI
3	alex	CHENNAI

In the above example, both the tables being joined have **ID** column(same name and same datatype), hence the records for which value of **ID** matches in both the tables will be the result of Natural Join of these two tables.

OUTER JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

LEFT Outer Join

The left outer join returns a resultset table with the **matched data** from the two tables and then the remaining rows of the **left** table and null from the **right** table's columns.

Syntax for Left Outer Join is,

```
SELECT column-name-list FROM

table-name1 LEFT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;
```

Copy

To specify a condition, we use the `ON` keyword with Outer Join.

Left outer Join Syntax for **Oracle** is,

```
SELECT column-name-list FROM

table-name1, table-name2 on table-name1.column-name = table-
name2.column-name(+);
```

Copy

Example of Left Outer Join

Here is the **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

5	ashish

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Left Outer Join query will be,

```
SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id =
class_info.id);
```

Copy

The resultset table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
4	anu	null	null
5	ashish	null	null

RIGHT Outer Join

The right outer join returns a resultset table with the **matched data** from the two tables being joined, then the remaining rows of the **right** table and null for the remaining **left** table's columns.

Syntax for Right Outer Join is,

```
SELECT column-name-list FROM

table-name1 RIGHT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;
```

Copy

Right outer Join Syntax for **Oracle** is,

```
SELECT column-name-list FROM

table-name1, table-name2

ON table-name1.column-name(+) = table-name2.column-name;
```

Copy

Example of Right Outer Join

Once again the **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	DELHI

2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Right Outer Join query will be,

```
SELECT * FROM class RIGHT OUTER JOIN class_info ON (class.id =
class_info.id);
```

Copy

The resultant table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
null	null	7	NOIDA
null	null	8	PANIPAT

Full Outer Join

The full outer join returns a resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Syntax of Full Outer Join is,

```
SELECT column-name-list FROM

table-name1 FULL OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;
```

Copy

Example of Full outer join is,

The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Full Outer Join query will be like,

```
SELECT * FROM class FULL OUTER JOIN class_info ON (class.id =
class_info.id);
```

Copy

The resultset table will look like,

ID	NAME	ID	Address

1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
4	anu	null	null
5	ashish	null	null
null	null	7	NOIDA
null	null	8	PANIPAT

Now that we have learned SQL JOIN, you can check these SQL topics as well and their usage:

- SQL function
- SQL Alias
- SQL SET operation
- SQL Views

SQL Alias - AS Keyword

Alias is used to give an alias name to a table or a column, which can be a resultset table too. This is quite useful in case of large or complex queries. Alias is mainly used for giving a short alias name for a column or a table with complex names.

Syntax of Alias for table names,

```
SELECT column-name FROM table-name AS alias-name
```

Copy

Following is an SQL query using **alias**,

```
SELECT * FROM Employee_detail AS ed;
```

Copy

Syntax for defining alias for columns will be like,

```
SELECT column-name AS alias-name FROM table-name;
```

Copy

Example using alias for columns,

```
SELECT customer_id AS cid FROM Emp;
```

Copy

Example of Alias in SQL Query

Consider the following two tables,

The **class** table,

ID	Name
1	abhi
2	adam
3	alex
4	anu
5	ashish

and the **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Below is the Query to fetch data from both the tables using SQL Alias,

```
SELECT C.id, C.Name, Ci.Address from Class AS C, Class_info AS Ci where
C.id = Ci.id;
```

Copy

and the resultset table will look like,

ID	Name	Address
1	abhi	DELHI
2	adam	MUMBAI
3	alex	CHENNAI

SQL Alias seems to be quite a simple feature of SQL, but it is highly useful when you are working with more than 3 tables and have to use JOIN on them.

SET Operations in SQL

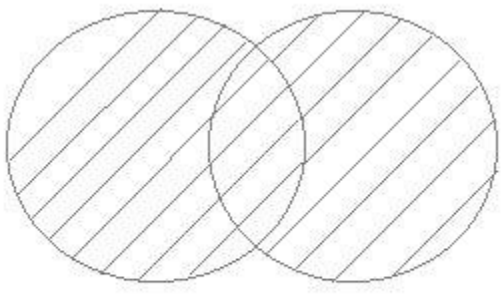
SQL supports few Set operations which can be performed on the table data. These are used to get meaningful results from data stored in the table, under different special conditions.

In this tutorial, we will cover 4 different types of SET operations, along with example:

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS

UNION Operation

UNION is used to combine the results of two or more `SELECT` statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.



Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

```
SELECT * FROM First

UNION

SELECT * FROM Second;
```

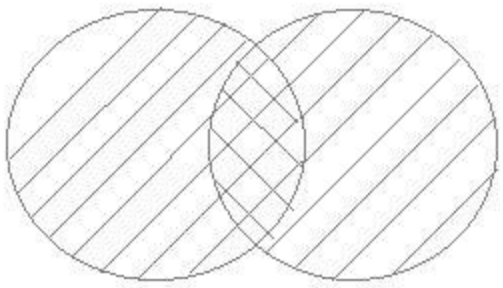
Copy

The resultset table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

UNION ALL

This operation is similar to Union. But it also shows the duplicate rows.



Example of Union All

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Union All query will be like,

```
SELECT * FROM First

UNION ALL

SELECT * FROM Second;
```

Copy

The resultset table will look like,

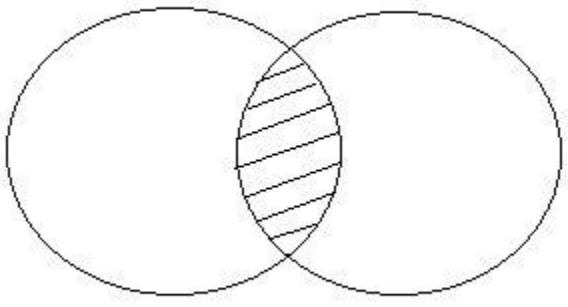
ID	NAME
1	abhi
2	adam

2	adam
3	Chester

INTERSECT

Intersect operation is used to combine two `SELECT` statements, but it only retuns the records which are common from both `SELECT` statements. In case of **Intersect** the number of columns and datatype must be same.

NOTE: MySQL does not support INTERSECT operator.



Example of Intersect

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

```
SELECT * FROM First

INTERSECT
```

```
SELECT * FROM Second;
```

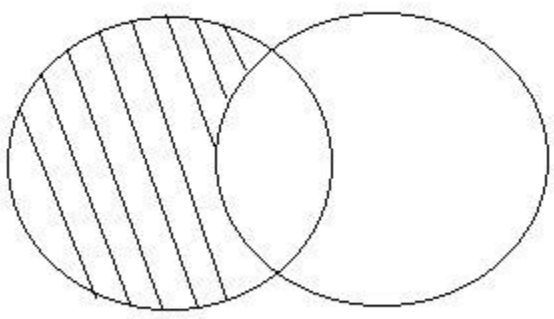
Copy

The resultset table will look like

ID	NAME
2	adam

MINUS

The Minus operation combines results of two `SELECT` statements and return only those in the final result, which belongs to the first set of the result.



Example of Minus

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Minus query will be,

```
SELECT * FROM First
```

```
MINUS

SELECT * FROM Second;
```

Copy

The resultset table will look like,

ID	NAME
1	abhi

What is an SQL Sequence?

Sequence is a feature supported by some database systems to produce unique values on demand. Some DBMS like **MySQL** supports `AUTO_INCREMENT` in place of Sequence.

`AUTO_INCREMENT` is applied on columns, it automatically increments the column value by `1` each time a new record is inserted into the table.

Sequence is also some what similar to `AUTO_INCREMENT` but it has some additional features too.

Creating a Sequence

Syntax to create a sequence is,

```
CREATE SEQUENCE sequence-name

    START WITH initial-value

    INCREMENT BY increment-value

    MAXVALUE maximum-value

    CYCLE | NOCYCLE;
```

Copy

- The **initial-value** specifies the starting value for the Sequence.
- The **increment-value** is the value by which sequence will be incremented.
- The **maximum-value** specifies the upper limit or the maximum value upto which sequence will increment itself.
- The keyword `CYCLE` specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the begining.
- And, `NO CYCLE` specifies that if sequence exceeds `MAXVALUE` value, an error will be thrown.

Using Sequence in SQL Query

Let's start by creating a sequence, which will start from 1, increment by 1 with a maximum value of 999.

```
CREATE SEQUENCE seq_1

START WITH 1

INCREMENT BY 1

MAXVALUE 999

CYCLE;
```

Copy

Now let's use the sequence that we just created above.

Below we have a **class** table,

ID	NAME
1	abhi
2	adam
4	alex

The SQL query will be,

```
INSERT INTO class VALUE(seq_1.nextval, 'anu');
```

Copy

Resultset table will look like,

ID	NAME
1	abhi
2	adam
4	alex
1	anu

Once you use nextval the sequence will increment even if you don't Insert any record into the table.

SQL VIEW

A VIEW in SQL is a logical subset of data from one or more tables. View is used to restrict data access.

Syntax for creating a View,

```
CREATE or REPLACE VIEW view_name

    AS

    SELECT column_name(s)

    FROM table_name

    WHERE condition
```

Copy

As you may have understood by seeing the above SQL query, a view is created using data fetched from some other table(s). It's more like a temporary table created with data.

Creating a VIEW

Consider following **Sale** table,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

SQL Query to Create a View from the above table will be,

```
CREATE or REPLACE VIEW sale_view

AS

SELECT * FROM Sale WHERE customer = 'Alex';
```

Copy

The data fetched from `SELECT` statement will be stored in another object called **sale_view**. We can use `CREATE` and `REPLACE` seperately too, but using both together works better, as if any view with the specified name exists, this query will replace it with fresh data.

Displaying a VIEW

The syntax for displaying the data in a view is similar to fetching data from a table using a `SELECT` statement.

```
SELECT * FROM sale_view;
```

Copy

Force VIEW Creation

`FORCE` keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATE or REPLACE FORCE VIEW view_name AS

    SELECT column_name(s)

    FROM table_name

    WHERE condition;
```

Copy

Update a VIEW

`UPDATE` command for view is same as for tables.

Syntax to Update a View is,

```
UPDATE view-name SET VALUE

WHERE condition;
```

Copy

NOTE: If we update a view it also updates base table data automatically.

Read-Only VIEW

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

```
CREATE or REPLACE FORCE VIEW view_name AS

    SELECT column_name(s)

    FROM table_name

    WHERE condition WITH read-only;
```

Copy

The above syntax will create view for **read-only** purpose, we cannot Update or Insert data into read-only view. It will throw an **error**.

Types of View

There are two types of view,

- Simple View
- Complex View

Simple View	Complex View
Created from one table	Created from one or more table
Does not contain functions	Contain functions
Does not contain groups of data	Contains groups of data