MODULE II

RELATIONAL DBMS

- 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.

BASIC CONCEPTS

Table:

- 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

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

Attribute:

- A table consists of several records(row), each record can be broken down into several smaller parts of data known as **Attributes**.
- In 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

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.

Relation instance:

- A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation Key:

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

<u>Relational Integrity Constraints</u>

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 refer 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 **Alice** is my girlfriend, then a girl with name Alice 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.

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.

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 extend 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

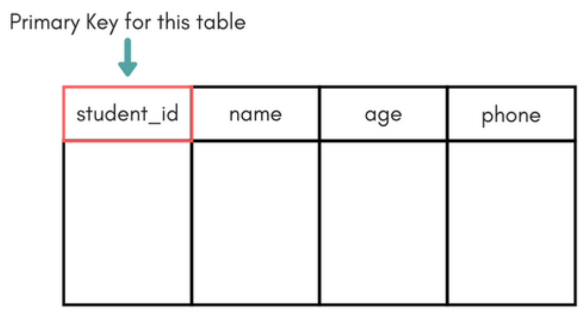
- **Super Key** is defined as a set of attributes within a table that can uniquely identify each record within a table. Super Key is a superset of Candidate key.
- In the table defined above super key would include student_id, (student_id, name), phone etc.
- The first one is pretty simple as student_id 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.

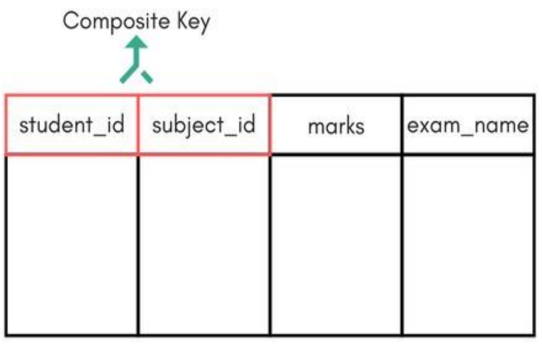
- In our example, student_id and phone both are candidate keys for table **Student**.
- A candiate key can never be NULL or empty. And its value should be unique.
- There can be more than one candidate keys for a table.
- A candidate key can be a combination of more than one columns(attributes).

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



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

- 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.
- 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).**

Basic Concepts of ER Model in DBMS

- 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.

ER Model: Entity and Entity Set

- An Entity is generally a real-world object which has characteristics and holds relationships in a DBMS.
- Example: For a **School Management Software**, we will have to store **Student** information, **Teacher** information, **Classes**, **Subjects** taught in each class etc. Considering this example, **Student** is an entity, **Teacher** is an entity, similarly, **Class**, **Subject** etc are also entities.
- 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

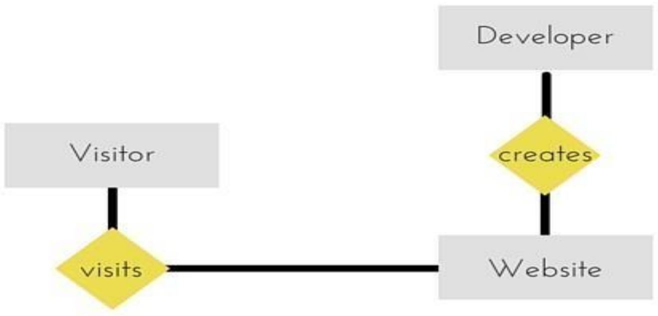
ER Model: Relationships

- 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, because 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.

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 given 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

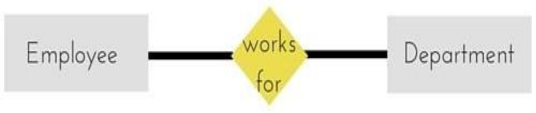
- Entitiy, Attributes, Relationships etc form the components of ER Diagram and there are defined symbols and shapes to represent each one of them.

Entity

- Simple rectangular box represents an Entity.

Student
Subject

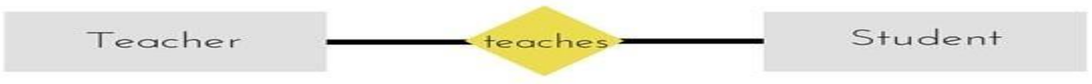
- An **Entity** can be any object, place, person, or class.
- In ER Diagram, an **entity** is represented using rectangles.
- Consider an example of an Organisation- Employee, Manager, Department, Product and many more can be taken as entities in an Organisation.



- The yellow rhombus in between represents a relationship.

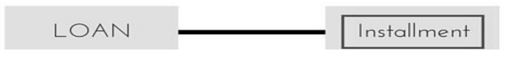
Relationships between Entities - Weak and Strong

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



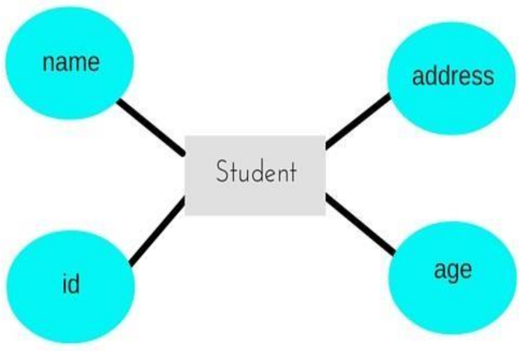
Weak Entity

- A weak Entity is represented using double rectangular boxes. It is generally connected to another entity.
- Weak entity is an entity that depends on another entity.
- Weak entity doesn't have any key attribute of its own.
- Double rectangle is used to represent a weak entity.



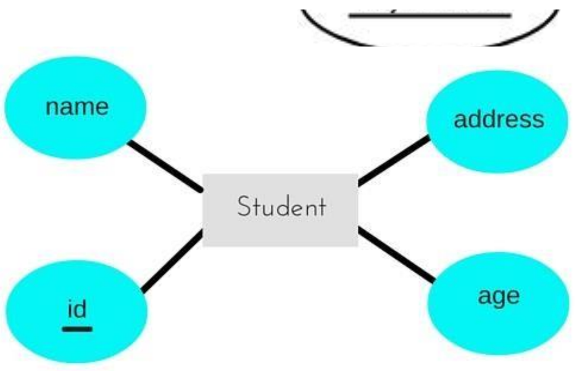
Attributes for any Entity

- Ellipse is used to represent attributes of any entity. It is connected to the entity.
- An **Attribute** describes a property or characteristic of an entity. For example, **Name**, **Age**, **Address** etc can be attributes of a **Student**.
- An attribute is represented using eclipse.



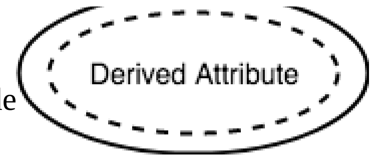
Key Attribute for any Entity

- To represent a Key attribute, the attribute name inside the Ellipse is underlined.
- Key attribute represents the main characteristic of an Entity. It is used to represent a Primary key. Ellipse with the text underlined, represents Key Attribute.



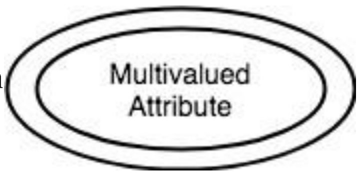
Derived Attribute for any Entity

- 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.



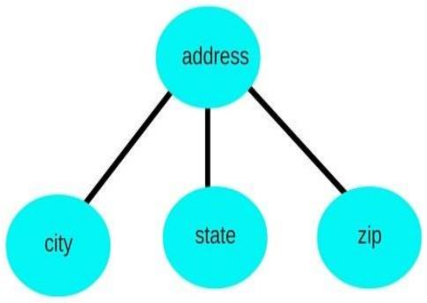
Multivalued Attribute for any Entity

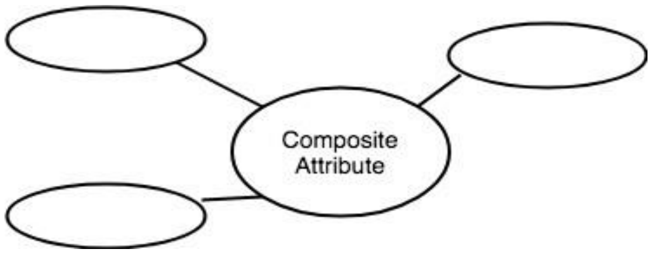
- Double Ellipse, one inside another, represents the attribute which can have multiple values.



Composite Attribute for any Entity

- 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.

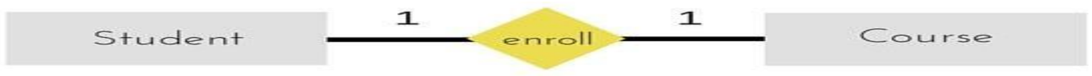
1. Binary Relationship
2. Recursive Relationship
3. Ternary Relationship

ER Diagram: Binary Relationship

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

1. One to One Relationship

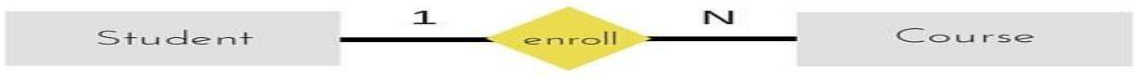
- This type of relationship is rarely seen in real world.



- The above example describes that one student can enroll only for one course and a course will also have only one Student.

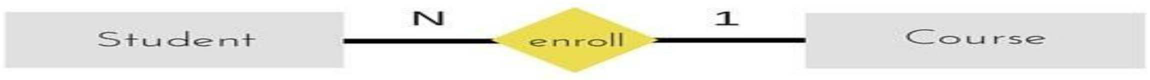
2. One to Many Relationship

- The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student.

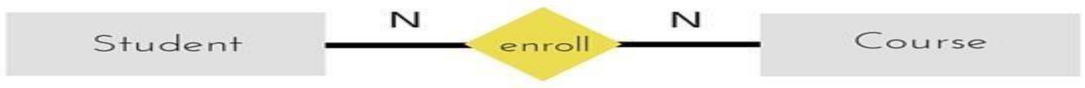


3. Many to One Relationship

- 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.



4. Many to Many Relationship



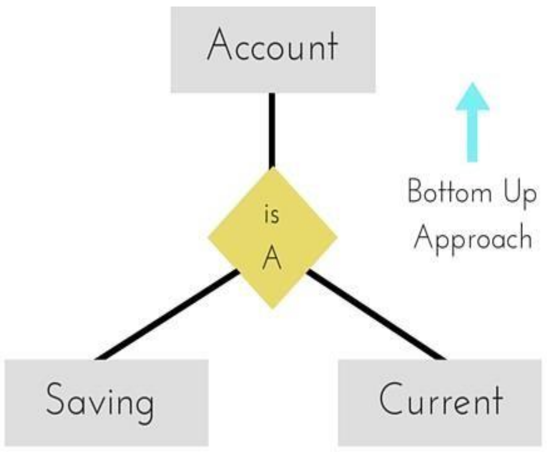
- The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.

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:

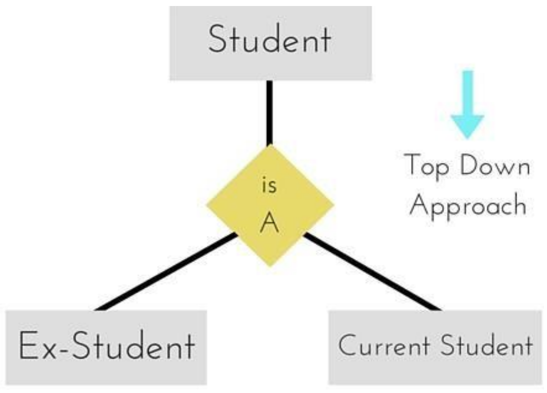
1. Generalization
2. Specialization
3. Aggregation

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

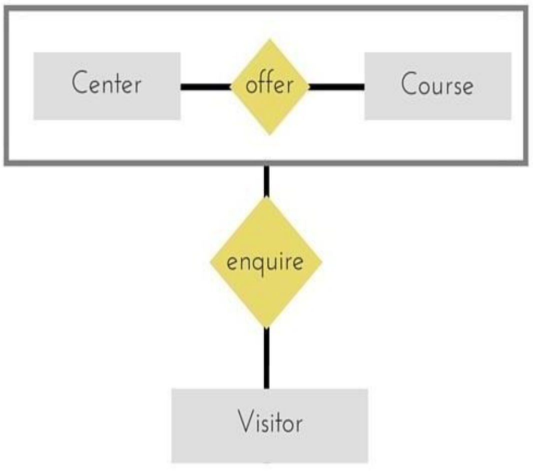


- **Specialization** is opposite to Generalization.
- It is a top-down approach in which one higher level entity can be broken down into two lower level entity.
- in specialization, a higher level entity may not have

any lower-level entity sets, it's possible.

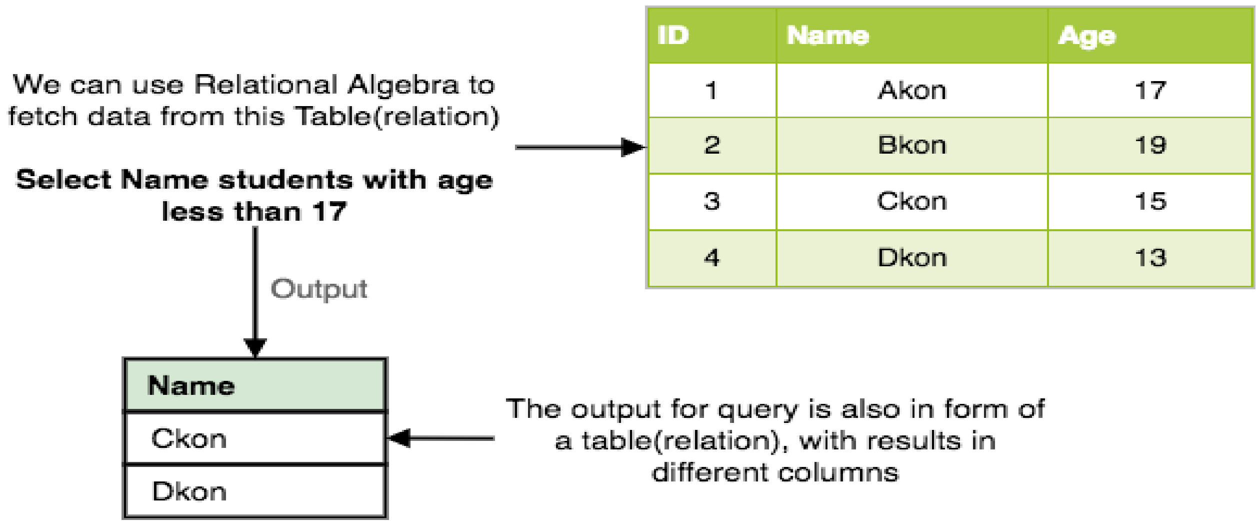
Aggregation

- Aggregation is a process when relation between two entities is treated as a **single entity**.
- In the diagram, 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.



Relational Algebra

- Every database management system must define a query language to allow users to access the data stored in the database.
- 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).
- It uses operators to perform queries.
- An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output.


We can use Relational Algebra to fetch data from this Table(relation)

ID	Name	Age
1	Akon	17
2	Bkon	19
3	Ckon	15
4	Dkon	13

Select Name students with age less than 17

Name
Ckon
Dkon

The output for query is also in form of a table(relation), with results in different columns

- Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

1. Select Operation (σ)

- It selects tuples that satisfy the given predicate from a relation.

Notation − $\sigma_p(r)$

σ - selection predicate

r - relation.

P - prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

For example − 1) $\sigma_{subject = "database"}$(Books)

Output − Selects tuples from books where subject is 'database'.

2) $\sigma_{\text{subject = "database" and price = "450"}}$(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450.

3) $\sigma_{\text{subject = "database" and price = "450" or year > "2010"}}$(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

2. Project Operation (∏)

- It projects column(s) that satisfy a given predicate.

Notation − $\prod_{A1, A2, An}$ (r)

Where A_1, A_2 , A_n are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

For example −

$\prod_{\text{subject, author}}$ (Books)

Selects and projects columns named as subject and author from the relation Books.

3. Union Operation (∪)

It performs binary union between two given relations and is defined as −

r ∪ s = { t | t ∈ r or t ∈ s}

Notation − r U s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

- **r**, and **s** must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

$\prod_{\text{author}}$ (Books) ∪ $\prod_{\text{author}}$ (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

4. Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − **r − s**

Finds all the tuples that are present in **r** but not in **s**.

$\prod_{\text{author}}$ (Books) − $\prod_{\text{author}}$ (Articles)

Output − Provides the name of authors who have written books but not articles.

5. Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where **r** and **s** are relations and their output will be defined as −

r X s = { q t | q ∈ r and t ∈ s}

$\sigma_{\text{author = 'tutorialspoint'}}$(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by tutorialspoint.

6. Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho** *ρ*.

Notation − ρ_{x} **(E)**

ρ(RelationNew, RelationOld)

Where the result of expression **E** is saved with name of **x**.

Additional operations are −

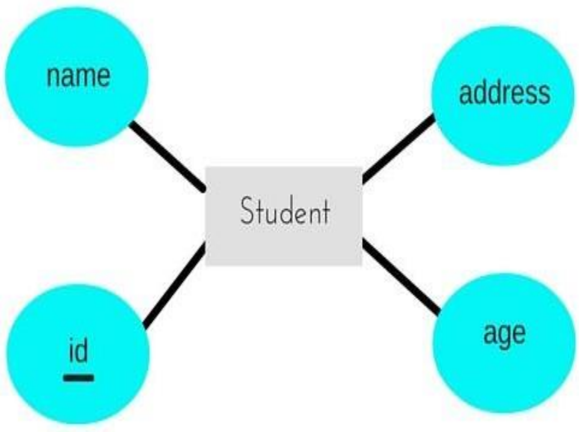
- Set intersection
- Assignment
- Natural join

ER Model to Relational Model

- ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand.
- ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram.
- Not all the ER Model constraints and components can be directly transformed into relational model, but an approximate schema can be derived.
- There are several processes and algorithms available to convert ER Diagrams into Relational Schema.

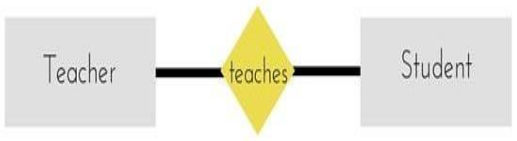
Entity becomes Table

- 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 given 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 represent 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.
- 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 foreign key constraints must be set for all the tables.



Note:

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.

MODULE III

Introduction to SQL

SQL is a standard language for accessing and manipulating databases.

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS to style the page

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.
- Look at the "Customers" table:

  Example : SELECT * FROM Customers;
- Every table is broken up into smaller entities called fields. The fields in the Customers table consist of CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country.
- A field is a column in a table that is designed to maintain specific information about every record in the table.
- A record, also called a row, is each individual entry that exists in a table. For example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

- A column is a vertical entity in a table that contains all information associated with a specific field in a table.

SQL Syntax

Database Tables

- A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.
- Here we will use the well-known Northwind sample database (included in MS Access and MS SQL Server).

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The table 3.1 above contains five records (one for each customer) and seven columns (CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country). This table will be used in all examples in this session.

SQL Statements

- Most of the actions you need to perform on a database are done with SQL statements.
- The following SQL statement selects all the records in the "Customers" table:
- Example: SELECT * FROM Customers;
- SQL keywords are NOT case sensitive: select is the same as SELECT

Semicolon after SQL Statements?

- Some database systems require a semicolon at the end of each SQL statement.
- Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

Some of The Most Important SQL Commands

- **SELECT** - extracts data from a database
- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database

- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database
- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

The SQL SELECT Statement

- The SELECT statement is used to select data from a database.
- The data returned is stored in a result table, called the result-set.

SELECT Syntax

```
SELECT column1, column2, ...
FROM table_name;
```

- Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:
- SELECT * FROM *table_name*;

SELECT Column Example

- The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:
- **Example:** SELECT CustomerName, City FROM Customers;

SELECT * Example

- The following SQL statement selects all the columns from the "Customers" table:
- **Example:** SELECT * FROM Customers;

The SQL SELECT DISTINCT Statement

- The SELECT DISTINCT statement is used to return only distinct (different) values.
- Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.
- The SELECT DISTINCT statement is used to return only distinct (different) values.

SELECT DISTINCT Syntax

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

SELECT Example

- The following SQL statement selects all (and duplicate) values from the "Country" column in the "Customers" table:

- **Example:** SELECT Country FROM Customers;

SELECT DISTINCT Examples

- The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

- **Example:** SELECT DISTINCT Country FROM Customers;

SELECT COUNT

- The following SQL statement lists the number of different (distinct) customer countries:

  Example

  SELECT COUNT(DISTINCT Country) FROM Customers;

<u>The SQL WHERE Clause</u>

- The WHERE clause is used to filter records.
- The WHERE clause is used to extract only those records that fulfill a specified condition.

WHERE Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

WHERE Clause Example

- The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

- **Example :** SELECT * FROM Customers
  WHERE Country='Mexico';

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern

IN	To specify multiple possible values for a column

- **Example:**SELECT * FROM Customers
  WHERE CustomerID=1;

<u>The SQL AND, OR and NOT Operators</u>

- The WHERE clause can be combined with AND, OR, and NOT operators.
- The AND and OR operators are used to filter records based on more than one condition:
  - The AND operator displays a record if all the conditions separated by AND is TRUE.
  - The OR operator displays a record if any of the conditions separated by OR is TRUE.
- The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

OR Syntax

NOT Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

AND Example

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

Example : SELECT * FROM Customers
WHERE Country='Germany' AND City='Berlin';

OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "München":

Example : SELECT * FROM Customers
WHERE City='Berlin' OR City='München';

NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT

"Germany":

Example :
```
SELECT * FROM Customers
WHERE NOT Country='Germany';
```

Combining AND, OR and NOT

You can also combine the AND, OR and NOT operators.

- The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

Example 1:
```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München');
```

- The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

Example 2:
```
SELECT * FROM Customers
WHERE NOT Country='Germany' AND NOT Country='USA';
```

<u>The SQL ORDER BY Keyword</u>

- The ORDER BY keyword is used to sort the result-set in ascending or descending order.
- The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

ORDER BY Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

ORDER BY Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

Example

```
SELECT * FROM Customers
ORDER BY Country;
```

ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

Example

```
SELECT * FROM Customers
ORDER BY Country DESC;
```

ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column:

Example

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

<u>The SQL INSERT INTO Statement</u>

- The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

- It is possible to write the INSERT INTO statement in two ways.

1. The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode,
Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

The CustomerID column is an auto-increment field and will be generated automatically when a new record is inserted into the table.

Insert Data Only in Specified Columns

- It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

Example

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	null	null	Stavanger	null	Norway

What is a NULL Value?

- A field with a NULL value is a field with no value.
- If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: It is very important to understand that a NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

- Assume we have the following "Persons" table:

ID	LastName	FirstName	Address	City
1	Doe	John	542 W. 27th Street	New York
2	Bloggs	Joe		London
3	Roe	Jane		New York
4	Smith	John	110 Bishopsgate	London

Suppose that the "Address" column in the "Persons" table is optional. If a record is inserted with no value for "Address", the "Address" column will be saved with a NULL value.

- The following SQL statement uses the IS NULL operator to list all persons that have no address:

```
SELECT LastName, FirstName, Address FROM Persons
WHERE Address IS NULL;
```

The result-set will look like this:

LastName	FirstName	Address
Bloggs	Joe	
Roe	Jane	

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

<u>The SQL UPDATE Statement</u>

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Note: Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

UPDATE Table

- The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

UPDATE Multiple Records

- It is the WHERE clause that determines how many records that will be updated.
- The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

Example

```
UPDATE Customers
SET ContactName='Juan'
WHERE Country='Mexico';
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Juan	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Juan	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

Example

```
UPDATE Customers
SET ContactName='Juan';
```

<u>The SQL DELETE Statement</u>

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

```
DELETE FROM table_name
WHERE condition;
```

Note: Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) that should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

Example

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```

The "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name;
```

or

```
DELETE * FROM table_name;
```

The SQL SELECT TOP Clause

- The SELECT TOP clause is used to specify the number of records to return.
- The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.
- **Note:** Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

SQL Server / MS Access Syntax:

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

MySQL Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
```

```
LIMIT number;
```

Oracle Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

SQL TOP, LIMIT and ROWNUM Examples

The following SQL statement selects the first three records from the "Customers" table:

Example

```
SELECT TOP 3 * FROM Customers;
```

The following SQL statement shows the equivalent example using the LIMIT clause:

Example

```
SELECT * FROM Customers
LIMIT 3;
```

The following SQL statement shows the equivalent example using ROWNUM:

Example

```
SELECT * FROM Customers
WHERE ROWNUM <= 3;
```

SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table:

Example

```
SELECT TOP 50 PERCENT * FROM Customers;
```

The SQL MIN() and MAX() Functions

- The MIN() function returns the smallest value of the selected column.
- The MAX() function returns the largest value of the selected column.

MIN() Syntax

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

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

MAX() Syntax

Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

MIN() Example

The following SQL statement finds the price of the cheapest product:

Example

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

MAX() Example

The following SQL statement finds the price of the most expensive product:

Example

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

<u>The SQL COUNT(), AVG() and SUM() Functions</u>

- The COUNT() function returns the number of rows that matches a specified criteria.
- The AVG() function returns the average value of a numeric column.
- The SUM() function returns the total sum of a numeric column.

COUNT() Syntax

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

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

AVG() Syntax

SUM() Syntax

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

Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

COUNT() Example

The following SQL statement finds the number of products:

Example

```
SELECT COUNT(ProductID)
FROM Products;
```

AVG() Example

The following SQL statement finds the average price of all products:

Example

```
SELECT AVG(Price)
FROM Products;
```

Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40

SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

Example

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

The SQL GROUP BY Statement

- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a new SQL database.

Syntax

```
CREATE DATABASE databasename;
```

CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

Example

```
CREATE DATABASE testDB;
```

The SQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

Syntax

```
DROP DATABASE databasename;
```

Note: Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

DROP DATABASE Example

The following SQL statement drops the existing database "testDB":

Example

```
DROP DATABASE testDB;
```

The SQL CREATE TABLE Statement

The CREATE TABLE statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

- The column parameters specify the names of the columns of the table.

- The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

Example

```
CREATE TABLE Persons (
   PersonID int,
   LastName varchar(255),
   FirstName varchar(255),
   Address varchar(255),
   City varchar(255)
);
```

- The PersonID column is of type int and will hold an integer.
- The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.
- The empty "Persons" table will now look like this:

PersonID	LastName	FirstName	Address	City

The empty "Persons" table can now be filled with data with the SQL INSERT INTO statement.

The SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

Syntax

```
DROP TABLE table_name;
```

Note: Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

Example

```
DROP TABLE Shippers;
```

SQL TRUNCATE TABLE Statement

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

Syntax

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype;
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

SQL ALTER TABLE Example

Look at the "Persons" table:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to add a column named "DateOfBirth" in the "Persons" table.

- We use the following SQL statement:

ALTER TABLE Persons
ADD DateOfBirth date;

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City	DateOfBirth
1	Hansen	Ola	Timoteivn 10	Sandnes	
2	Svendson	Tove	Borgvn 23	Sandnes	
3	Pettersen	Kari	Storgt 20	Stavanger	

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

and it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table

SQL INNER JOIN KEYWORD

The INNER JOIN keyword selects records that have matching values in both tables.

INNER JOIN Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.

Note: FULL OUTER JOIN can potentially return very large result-sets!

FULL OUTER JOIN Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2 ON table1.column_name = table2.column_name;
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
```

```
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

A selection from the result set may look like this:

CustomerName	OrderID
Alfreds Futterkiste	
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365
	10382
	10351

Note: The FULL OUTER JOIN keyword returns all the rows from the left table (Customers), and all the rows from the right table (Orders). If there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

SQL VIEWS

1. SQL CREATE VIEW Statement

- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

SQL CREATE VIEW Examples

- If you have the Northwind database you can see that it has several views installed by default.
- The view "Current Product List" lists all active products (products that are not discontinued) from the "Products" table. The view is created with the following SQL:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID, ProductName
FROM Products
WHERE Discontinued = No;
```

- Then, we can query the view as follows:

```
SELECT * FROM [Current Product List];
```

- Another view in the Northwind sample database selects every product in the "Products" table with a unit price higher than the average unit price:

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products);
```

- We can query the view above as follows:

```
SELECT * FROM [Products Above Average Price];
```

- Another view in the Northwind database calculates the total sale for each category in 1997. Note that this view selects its data from another view called "Product Sales for 1997":

```
CREATE VIEW [Category Sales For 1997] AS
SELECT DISTINCT CategoryName, Sum(ProductSales) AS CategorySales
FROM [Product Sales for 1997]
GROUP BY CategoryName;
```

- We can query the view above as follows:

```
SELECT * FROM [Category Sales For 1997];
```

- We can also add a condition to the query. Let's see the total sale only for the category "Beverages":

```
SELECT * FROM [Category Sales For 1997]
WHERE CategoryName = 'Beverages';
```

2. SQL Updating a View

You can update a view by using the following syntax:

SQL CREATE OR REPLACE VIEW Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Now we want to add the "Category" column to the "Current Product List" view. We will update the view with the following SQL:

```
CREATE OR REPLACE VIEW [Current Product List] AS
SELECT ProductID, ProductName, Category
FROM Products
WHERE Discontinued = No;
```

3. SQL Dropping a View

You can delete a view with the DROP VIEW command.

SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

<u>DBMS- TRANSACTION</u>

- A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

- Let’s take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

A’s Account

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

B’s Account

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

<u>ACID Properties</u>

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Serializability

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

Schedule − A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

Serial Schedule − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

- In a multi-transaction environment, serial schedules are considered as a benchmark.
- The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion.
- This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state.
- To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

Equivalence Schedules

An equivalence schedule can be of the following types −

Result Equivalence

If two schedules produce the same result after execution, they are said to be result equivalent. They may yield the same result for some value and different results for another set of values. That's why this equivalence is not generally considered significant.

View Equivalence

Two schedules would be view equivalence if the transactions in both the schedules perform similar actions in a similar manner.

For example −

- If T reads the initial data in S1, then it also reads the initial data in S2.
- If T reads the value written by J in S1, then it also reads the value written by J in S2.
- If T performs the final write on the data value in S1, then it also performs the final write on the data value in S2.

Conflict Equivalence

Two schedules would be conflicting if they have the following properties −

- Both belong to separate transactions.
- Both accesses the same data item.
- At least one of them is "write" operation.

Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if −

- Both the schedules contain the same set of Transactions.

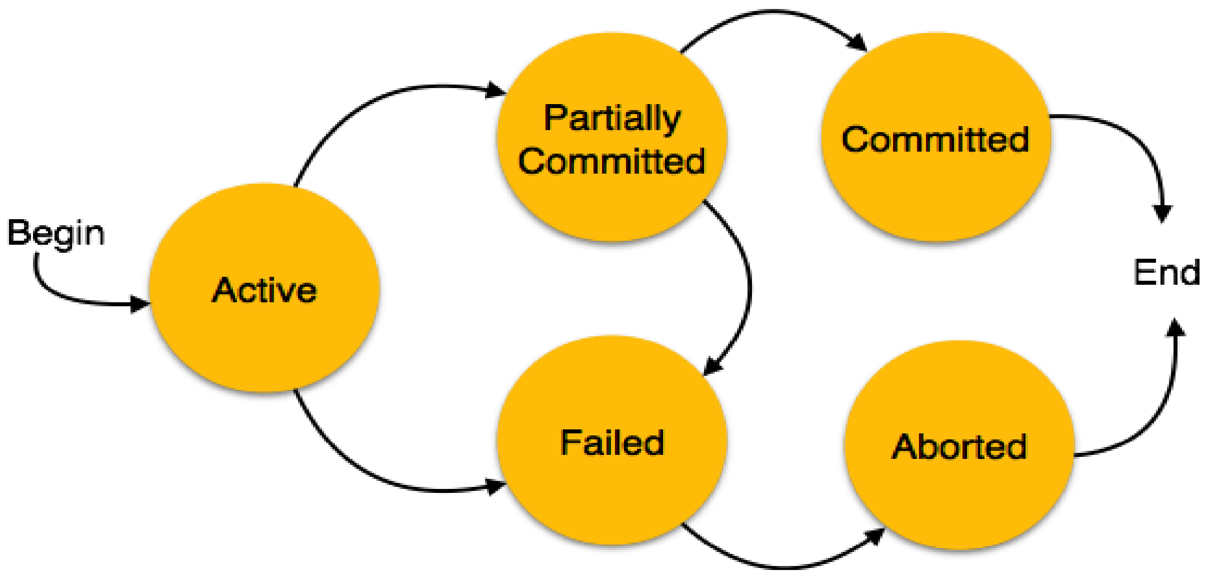
- The order of conflicting pairs of operation is maintained in both the schedules.

Note − View equivalent schedules are view serializable and conflict equivalent schedules are conflict serializable. All conflict serializable schedules are view serializable too.

States of Transactions

A transaction in a database can be in one of the following states −

- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.



- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
    - Re-start the transaction
    - Kill the transaction
- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

<u>TCL Commands- Commit, Rollback and Savepoint SQL commands</u>

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

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;
```

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;
```

- 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;
```

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 use the ROLLBACK command to roll back the state of data to the **savepoint B**.

```
ROLLBACK TO B;

SELECT * FROM class;
```

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;
```

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.

DATABASE CONNECTIVITY USING JDBC/ODBC

- **JDBC**: establishes a connection **with** a **database** sends SQL statements processes the results.
- **ODBC** is used between applications **JDBC** is used by Java programmers to connect to **databases With** a small "bridge" program, you can **use** the **JDBC** interface to **access ODBC**- accessible **databases**.

<u>ODBC</u>

- ODBC is (Open Database Connectivity): A standard or open application programming interface (API) for accessing a database.
- By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, DB2, Excel, and Text.
- It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases.
- ODBC handles the SQL request and converts it into a request the individual database system understands.

<u>JDBC</u>

- JDBC is: Java Database Connectivity is a Java API for connecting programs written in Java to the data in relational databases
- Consists of a set of classes and interfaces written in the Java programming language.
- It provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.
- The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.
    - establishes a connection with a database
    - sends SQL statements
    - processes the results.

<u>JDBC vs ODBC</u>

- ODBC is used between applications
- JDBC is used by Java programmers to connect to databases
- With a small "bridge" program, you can use the JDBC interface to access ODBC- accessible databases.

- JDBC allows SQL-based database access for EJB persistence and for direct manipulation from CORBA, DJB or other server objects .

MODULE IV

DBMS – Normalization

Functional Dependency

Functional dependency (FD) is a set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An, then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by an arrow sign (→) that is, X→Y, where X functionally determines Y. The left-hand side attributes determine the values of attributes on the right-hand side.

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

- **Reflexive rule** − If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule** − If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule** − Same as transitive rule in algebra, if a → b holds and b → c holds, then a → c also holds. a → b is called as a functionally that determines b.

Trivial Functional Dependency

- **Trivial** − If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
- **Non-trivial** − If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.
- **Completely non-trivial** − If an FD X → Y holds, where x intersect Y = Φ, it is said to be a completely non-trivial FD.

Normalization

- atabase 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 Anamolies.

- It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.
- **Update anomalies** − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
- **Deletion anomalies** − We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
- **Insert anomalies** − We tried to insert data in a record that does not exist at all.
- Normalization is a method to remove all these anomalies and bring the database to a consistent state. Normalization is used for mainly two purposes,

1. Eliminating reduntant(useless) data.
2. Ensuring data dependencies make sense i.e data is logically stored.

<u>First Normal Form (1NF)</u>

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.

- First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.
- We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

- Each attribute must contain only a single value from its pre-defined domain.

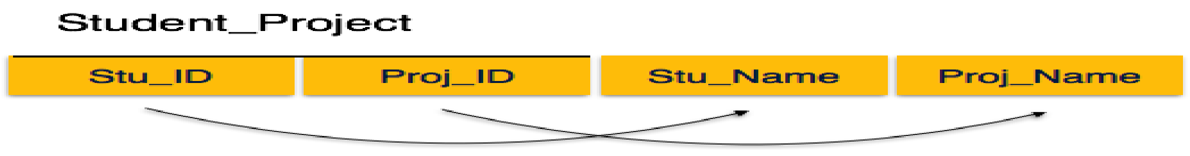
<u>Second Normal Form (2NF)</u>

For a table to be in the Second Normal Form,

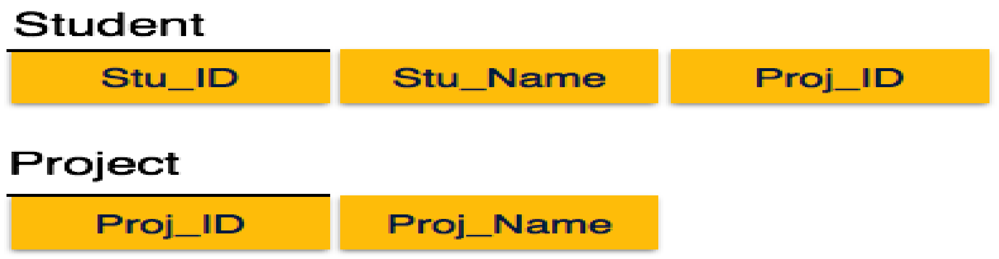
1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

- **Prime attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.

- **Non-prime attribute** −An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.
- If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute.
- That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true i.e, there should not be any partial dependancy.



- We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID.
- According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually.
- But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called **partial dependency**, which is not allowed in Second



Normal Form.

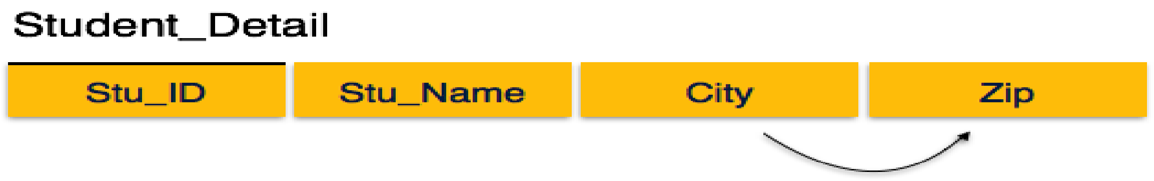
Third Normal Form (3NF)

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.

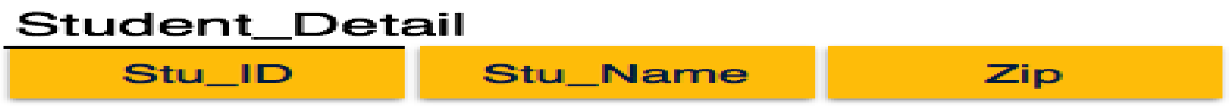
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy −

- No non-prime attribute is transitively dependent on prime key attribute.
- For any non-trivial functional dependency, X → A, then either −
  - X is a superkey or,
  - A is prime attribute.



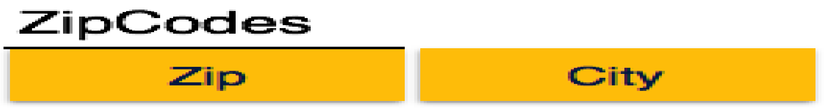
- We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists **transitive dependency**.

- To bring this relation into third normal form, we break the relation into two relations as follows:



Student_Detail

Stu_ID	Stu_Name	Zip



ZipCodes

Zip	City

<u>**Boyce and Codd Normal Form (BCNF)**</u>

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.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

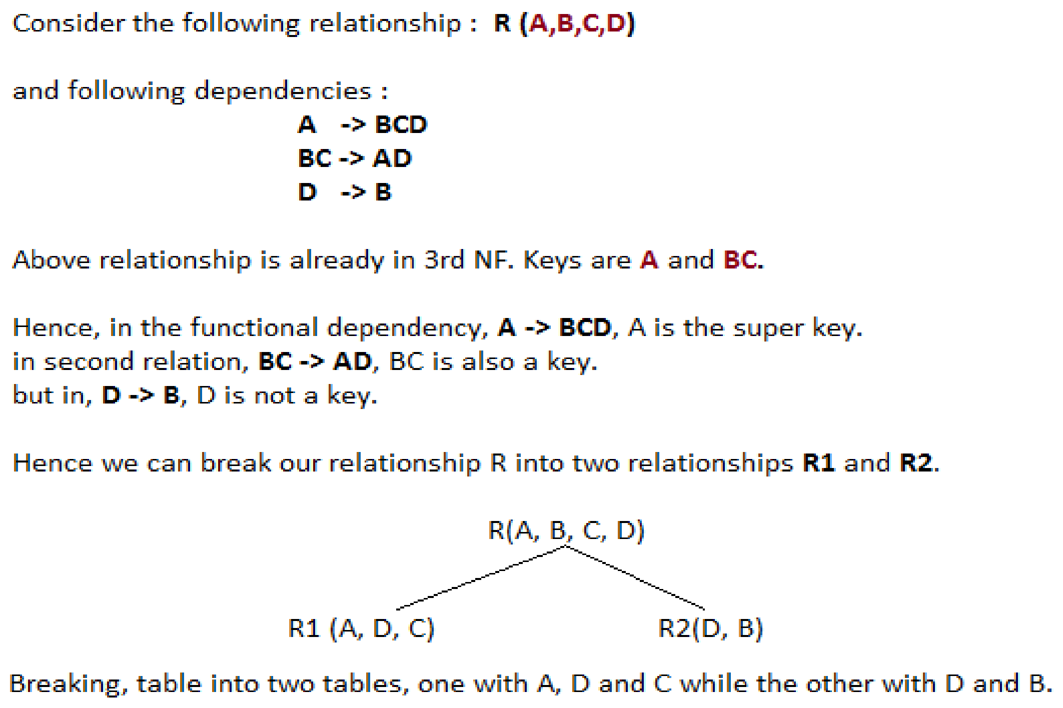
Stu_ID → Stu_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF.

Example of BCNF in terms of relations:



Consider the following relationship : **R (A,B,C,D)**

and following dependencies :

A -> BCD
BC -> AD
D -> B

Above relationship is already in 3rd NF. Keys are **A** and **BC.**

Hence, in the functional dependency, **A -> BCD**, A is the super key.
in second relation, **BC -> AD**, BC is also a key.
but in, **D -> B**, D is not a key.

Hence we can break our relationship R into two relationships **R1** and **R2**.

$$\begin{tikzcd}
 & R(A, B, C, D) \arrow[dl, dash] \arrow[dr, dash] & \\
R1 (A, D, C) & & R2(D, B)
\end{tikzcd}$$

Breaking, table into two tables, one with A, D and C while the other with D and B.

Fourth Normal Form (4NF)

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?

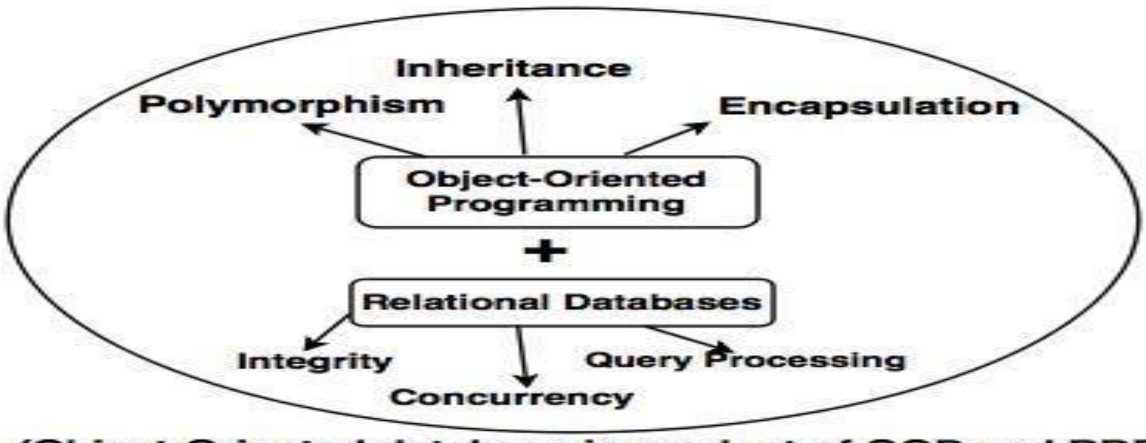
A table is said to have multi-valued dependency, if the following conditions are true,

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.

Object-Oriented Database Management System (OODBMS)

- An object-oriented database management system (OODBMS) is a database management system that supports the creation and modeling of data as objects.
- OODBMS also includes support for classes of objects and the inheritance of class properties, and incorporates methods, subclasses and their objects.
- Most of the object databases also offer some kind of query language, permitting objects to be found through a declarative programming approach.
- Also called an object database management system (ODMS)



(Object-Oriented database is product of OOP and RDB)

Object-Oriented database

Features of OODBMS

In OODBMS, every entity is considered as object and represented in a table. Similar objects are classified to classes and subclasses and relationship between two object is maintained using concept of inverse reference.

Some of the features of OODBMS are as follows:

1. Complexity
OODBMS has the ability to represent the complex internal structure (of object) with multilevel complexity.

2. Inheritance
Creating a new object from an existing object in such a way that new object inherits all characteristics of an existing object.

3. Encapsulation
It is an data hiding concept in OOPL which binds the data and functions together which can manipulate data and not visible to outside world.

4. Persistency
OODBMS allows to create persistent object (Object remains in memory even after execution). This feature can automatically solve the problem of recovery and concurrency.

Challenges in ORDBMS implementation

During the implementation of ORDBMS, various challenges arise which need to be resolved. They are:

1. Storage and accessibility of data
It is possible to define new types with new access to structures with the help of OODBMS. Hence, it is important that the system must store ADT and structured objects efficiently along with the provision of indexed access.
Challenge : Storage of large ADTs and structured objects.
Solution: As large ADTs need special storage, it is possible to store them on different locations on the disk from the tuples that contain them. **For e.g.** BLOBs (Binary Large Object like images, audio or any multimedia object.)
Use of flexible disk layout mechanisms can solve the storage problem of structured objects.

2. Query Processing

Challenge: Efficient flow of Query Processing and optimization is a difficult task.
Solution: By registering the user defined aggregation function, query processing becomes easier. It requires three implementation steps - **initialize, iterate and terminate.**

3. Query Optimization

Challenge: New indexes and query processing techniques increase the options for query optimization. But, the challenge is that the optimizer must know to handle and use the query processing functionality properly.
Solution: While constructing a query plan, an optimizer must be familiar to the newly added index structures

Introduction to Parallel Databases

Companies need to handle huge amount of data with high data transfer rate. The client server and centralized system is not much efficient. The need to improve the efficiency gave birth to the concept of Parallel Databases.

Parallel database system improves performance of data processing using multiple resources in parallel, like multiple CPU and disks are used parallely.

It also performs many parallelization operations like, data loading and query processing.

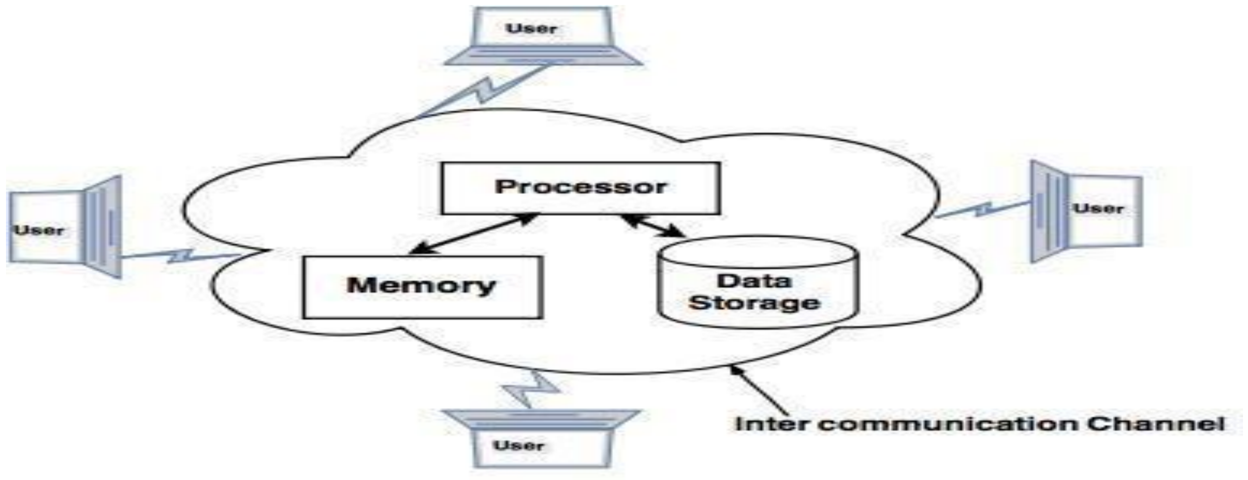
Goals of Parallel Databases

The concept of Parallel Database was built with a goal to:

Improve performance:
The performance of the system can be improved by connecting multiple CPU and disks in parallel. Many small processors can also be connected in parallel.

Improve availability of data:
Data can be copied to multiple locations to improve the availability of data.
For example: if a module contains a relation (table in database) which is unavailable then it is important to make it available from another module.

Improve reliability:
Reliability of system is improved with completeness, accuracy and availability of data.



Parallel database system

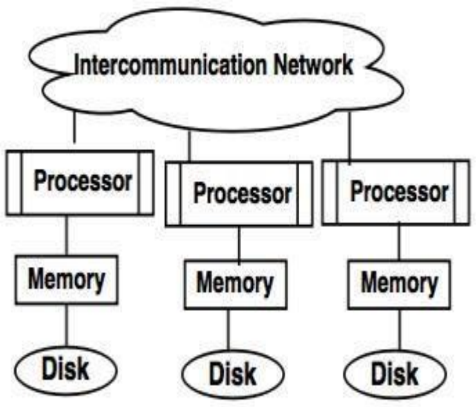
Provide distributed access of data:
Companies having many branches in multiple cities can access data with the help of parallel database system.

<u>**Architecture**</u>
There are three main architectures that have been proposed for parallel database management systems :

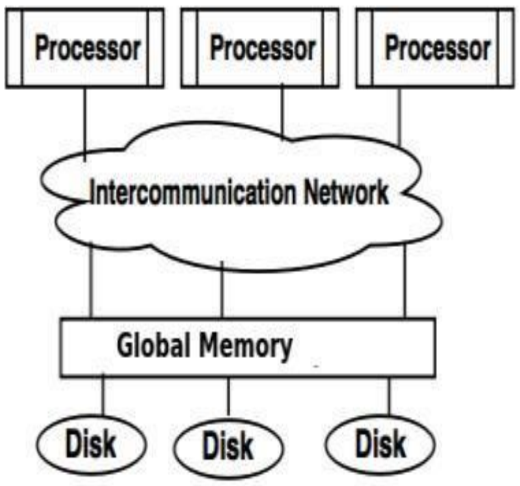
- **Shared Memory** – Any CPU has access to both memory and disk through a fast interconnect (e.g high-speed bus). This provides excellent load balance however scalability and availability is limited.
- **Shared Disk** – This provides the CPU with its own memory but a shared disk. Meaning there is no longer competition for shared memory but still competition for access to the

shared disk. This provides better scale up and the load balancing is still acceptable. Availability is better than shared memory but still limited as disk failure would mean entire system failure.

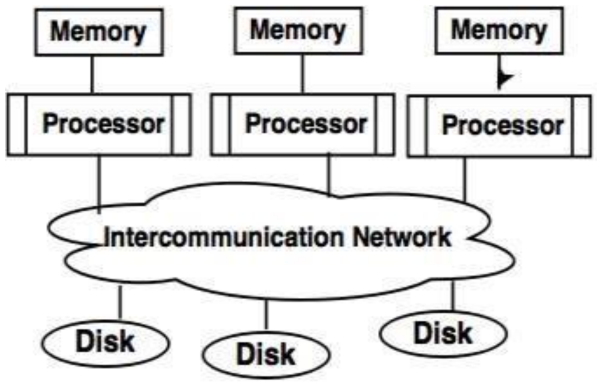
- **Share Nothing** – Each processor has exclusive access to its main memory and disk unit, this means all communication between CPUs is through a network connection. Shared nothing has high availability and reliability; if one node fails the others are still able to run independently. However load balance and skew become major issues with this architecture.



Shared nothing disk system in Parallel Databases



Shared Memory System in Parallel Databases



Shared disk system in Parallel Databases