**DBMS COMPLETE UNIT - 2**

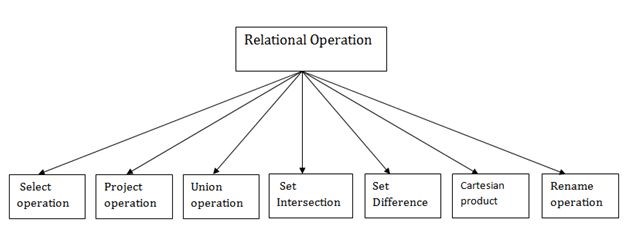
**For better understanding watch this youtube playlist along with the topics covered in this written material--**

<https://youtube.com/playlist?list=PLxCzCOWd7aiFAN6I8CuViBuCdJgiOkT2Y>

# Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

## Types of Relational operation



1. **Select Operation:** 
   * The select operation selects tuples that satisfy a given predicate.
   * It is denoted by sigma (σ).

1. Notation: σ p(r)

**Where:**

**σ**​ is used for selection prediction

**r**​ is used for relation

**p**​ is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relations can be used as relational operators like =, ≠, ≥, <, >, ≤.

**For example: LOAN Relation**

|  |  |  |
| --- | --- | --- |
| **BRANCH\_NAME** | **LOAN\_NO** | **AMOUNT** |
| Downtown | L-17 | 1000 |
| Redwood | L-23 | 2000 |
| Perryride | L-15 | 1500 |
| Downtown | L-14 | 1500 |
| Mianus | L-13 | 500 |
| Roundhill | L-11 | 900 |
| Perryride | L-16 | 1300 |

**Input:**

1. σ BRANCH\_NAME=​"perryride"​ (LOAN) **Output:**

|  |  |  |
| --- | --- | --- |
| **BRANCH\_NAME** | **LOAN\_NO** | **AMOUNT** |
| Perryride | L-15 | 1500 |
| Perryride | L-16 | 1300 |

1. **Project Operation:** 
   * This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
   * It is denoted by ∏.

1. Notation: ∏ A1, A2, An (r)

**Where**

**A1**​, ​**A2**​, ​**A3**​ is used as an attribute name of relation ​**r**​.

**Example: CUSTOMER RELATION**

|  |  |  |
| --- | --- | --- |
| **NAME** | **STREET** | **CITY** |
| Jones | Main | Harrison |
| Smith | North | Rye |
| Hays | Main | Harrison |
| Curry | North | Rye |
| Johnson | Alma | Brooklyn |
| Brooks | Senator | Brooklyn |

**Input:**

1. ∏ NAME, CITY (CUSTOMER)

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | | **CITY** | |
| Jones | | Harrison | |
| Smith | | Rye | |
| Hays | | Harrison | |
| Curry | | Rye | |
| Johnson | | Brooklyn | |
| Brooks | | Brooklyn | |

1. **Union Operation:** 
   * Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
   * It eliminates the duplicate tuples. It is denoted by ∪.

1. Notation: R ∪ S

A union operation must hold the following condition:

* R and S must have the attribute of the same number.
* Duplicate tuples are eliminated automatically.

**Example:**

**DEPOSITOR RELATION**

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | **ACCOUNT\_NO** |
| Johnson | A-101 |
| Smith | A-121 |
| Mayes | A-321 |
| Turner | A-176 |
| Johnson | A-273 |
| Jones | A-472 |
| Lindsay | A-284 |

**BORROW RELATION**

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | **LOAN\_NO** |
| Jones | L-17 |
| Smith | L-23 |
| Hayes | L-15 |
| Jackson | L-14 |
| Curry | L-93 |
| Smith | L-11 |
| Williams | L-17 |

**Input:**

1. ∏ CUSTOMER\_NAME (BORROW) ∪ ∏ CUSTOMER\_NAME (DEPOSITOR)

**Output:**

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | |
| Johnson | |
| Smith | |
| Hayes | |
| Turner | |
| Jones | |
| Lindsay | |
| Jackson | |
| Curry | |
| Williams | |
| Mayes | |

1. **Set Intersection:** 
   * Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
   * It is denoted by intersection ∩.

1. Notation: R ∩ S

**Example:** Using the above DEPOSITOR table and BORROW table​

**Input:**

1. ∏ CUSTOMER\_NAME (BORROW) ∩ ∏ CUSTOMER\_NAME (DEPOSITOR) **Output:**

|  |
| --- |
| **CUSTOMER\_NAME** |
| Smith |
| Jones |

1. **Set Difference:** 
   * Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.
   * It is denoted by intersection minus (-).

1. Notation: R - S

**Example:** Using the above DEPOSITOR table and BORROW table​

**Input:**

1. ∏ CUSTOMER\_NAME (BORROW) - ∏ CUSTOMER\_NAME (DEPOSITOR)

**Output:**

|  |
| --- |
| **CUSTOMER\_NAME** |
| Jackson |
| Hayes |
| Willians |
| Curry |

### 6. Cartesian product

* The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.
* It is denoted by X.

1. Notation: E X D

**Example:**

**EMPLOYEE**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_DEPT** |
| 1 | Smith | A |
| 2 | Harry | C |
| 3 | John | B |

**DEPARTMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **DEPT\_NO** | | **DEPT\_NAME** | |
| A | | Marketing | |
| B | | Sales | |
| C | | Legal | |

**Input:**

1. EMPLOYEE X DEPARTMENT **Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_DEPT** | **DEPT\_NO** | **DEPT\_NAME** |
| 1 | Smith | A | A | Marketing |
| 1 | Smith | A | B | Sales |
| 1 | Smith | A | C | Legal |
| 2 | Harry | C | A | Marketing |
| 2 | Harry | C | B | Sales |
| 2 | Harry | C | C | Legal |
| 3 | John | B | A | Marketing |
| 3 | John | B | B | Sales |
| 3 | John | B | C | Legal |

**7. Rename Operation:**

The rename operation is used to rename the output relation. It is denoted by ​**rho** (ρ).

**Example:**​ We can use the rename operator to rename STUDENT relation to STUDENT1.

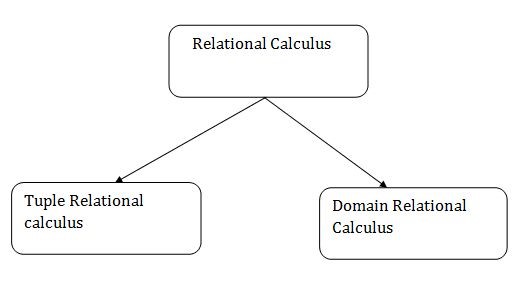
1. ρ(STUDENT1, STUDENT)

|  |
| --- |
| Note: Apart from these common operations Relational algebra can be used in Join |
| operations |

# Relational Calculus

* Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results.
* The relational calculus tells what to do but never explains how to do.

**Types of Relational calculus:**



## 1. Tuple Relational Calculus (TRC)

* The tuple relational calculus is specified to select the tuples in a relation. In

TRC, a filtering variable uses the tuples of a relation.

* The result of the relation can have one or more tuples.

**Notation:**

1. {T | P (T)} or {T | Condition (T)}

Where

**T** is the resulting tuples​

**P(T)** is the condition used to fetch T.​

**For example:**

1. { T.name | Author(T) AND T.article = 'database'​ }​

**OUTPUT:** This query selects the tuples from the AUTHOR relation. It returns a tuple​ with 'name' from Author who has written an article on 'database'.

TRC (tuple relational calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

**For example:**

1. { R| ∃T ∈ Authors(T.article='database'​ AND R.name=T.name)}​

**Output:** This query will yield the same result as the previous one.​

## 2. Domain Relational Calculus (DRC)

* The second form of relation is known as Domain relational calculus. In domain relational calculus, the filtering variable uses the domain of attributes.
* Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ┓ (not).
* It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.

**Notation:**

1. { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where

**a1, a2** are attributes​

**P** stands for formula built by inner attributes​

**For example:**

1. {< article, page, subject > | ∈ javatpoint ∧ subject = ​'database'​}

**Output:**​ This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database

# SQL

**SQL**​ (​*Structured Query Language*​) is used to perform operations on the records stored in the database such as updating records, deleting records, creating and modifying tables, views, etc.

SQL is just a query language; it is not a database. To perform SQL queries, you need to install any database, for example, Oracle, MySQL, MongoDB, PostGreSQL, SQL Server, DB2, etc.

## What is SQL

* SQL stands for ​**Structured Query Language**​.
* It is designed for managing data in a relational database management system (RDBMS).
* It is pronounced as S-Q-L or sometimes ​**See-Qwell**​.
* SQL is a database language, it is used for database creation, deletion, fetching rows, and modifying rows, etc.
* SQL is based on relational algebra and tuple relational calculus.

All DBMS like MySQL, Oracle, MS Access, Sybase, Informix, PostgreSQL, and SQL Server use SQL as standard database language.

## Why SQL is required

SQL is required:

* To create new databases, tables and views
* To insert records in a database
* To update records in a database
* To delete records from a database
* To retrieve data from a database

## What SQL does

* With SQL, we can query our database in several ways, using English-like statements.
* With SQL, a user can access data from a relational database management system.
* It allows the user to describe the data.
* It allows the user to define the data in the database and manipulate it when needed.
* It allows the user to create and drop databases and tables.
* It allows the user to create a view, stored procedure, function in a database.
* It allows the user to set permission on tables, procedures, and views.

# SQL Commands

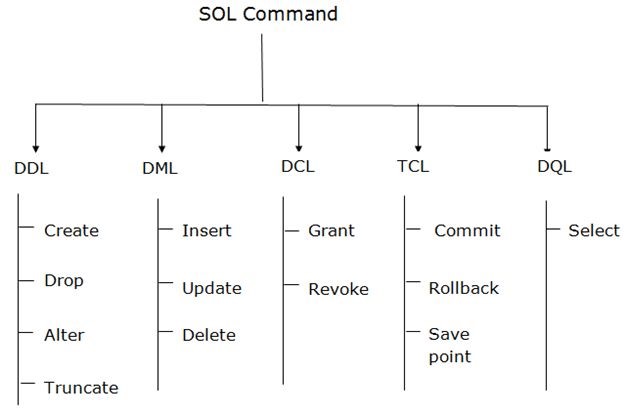
* SQL commands are instructions. It is used to communicate with the database.

It is also used to perform specific tasks, functions, and queries of data.

* SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

## Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



### 1. Data Definition Language (DDL)

* DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
* All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

* CREATE
* ALTER
* DROP
* TRUNCATE

1. **CREATE** It is used to create a new table in the database.​

**Syntax:**

* 1. CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[,....]);

**Example:**

1. CREATE TABLE EMPLOYEE(Name VARCHAR2(20​ )​ , Email VARCHAR2(100​ )​ ,

DOB DATE);

1. **DROP:** It is used to delete both the structure and record stored in the table.​

**Syntax**

* 1. DROP TABLE ;

**Example**

1. DROP TABLE EMPLOYEE;

1. **ALTER:** It is used to alter the structure of the database. This change could be​ either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

To add a new column in the table

* 1. ALTER TABLE table\_name ADD column\_name COLUMN-definition;

To modify existing column in the table:

1. ALTER TABLE MODIFY(COLUMN DEFINITION....);

**EXAMPLE**

* 1. ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20​ ))​ ;
  2. ALTER TABLE STU\_DETAILS MODIFY (NAME VARCHAR2(20​ ))​ ;

1. **TRUNCATE:** It is used to delete all the rows from the table and free the space​ containing the table.

**Syntax:**

* 1. TRUNCATE TABLE table\_name;

**Example:**

1. TRUNCATE TABLE EMPLOYEE;

### 2. Data Manipulation Language

* DML commands are used to modify the database. It is responsible for all form of changes in the database.
* The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rolled back.

Here are some commands that come under DML:

* INSERT
* UPDATE
* DELETE

1. **INSERT:** The INSERT statement is a SQL query. It is used to insert data into the​ row of a table.

**Syntax:**

* 1. INSERT INTO TABLE\_NAME
  2. (col1, col2, col3,.... col N)
  3. VALUES (value1, value2, value3, .... valueN);

Or

* 1. INSERT INTO TABLE\_NAME
  2. VALUES (value1, value2, value3, .... valueN);

**For example:**

1. INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo"​ , ​ "DBMS"​ )​ ;

1. **UPDATE:** This command is used to update or modify the value of a column in the​ table.

**Syntax:**

* 1. UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**For example:**

* 1. UPDATE students
  2. SET User\_Name = 'Sonoo'​
  3. WHERE Student\_Id = '3'​

1. **DELETE:** It is used to remove one or more row from a table.​

**Syntax:**

* 1. DELETE FROM table\_name [WHERE condition];

**For example:**

* 1. DELETE FROM javatpoint
  2. WHERE Author="Sonoo"​ ;​

### 3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

* Grant
* Revoke

1. **Grant:** It is used to give user access privileges to a database.​

**Example**

* 1. GRANT SELECT, UPDATE ON MY\_TABLE TO SOME\_USER, ANOTHER\_USER;

1. **Revoke:** It is used to take back permissions from the user.​

**Example**

* 1. REVOKE SELECT, UPDATE ON MY\_TABLE FROM USER1, USER2;

### 4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

* COMMIT
* ROLLBACK
* SAVEPOINT

1. **Commit:** Commit command is used to save all the transactions to the database.​

**Syntax:**

* 1. COMMIT;

**Example:**

* 1. DELETE FROM CUSTOMERS
  2. WHERE AGE = 25​ ;​
  3. COMMIT;

1. **Rollback:** Rollback command is used to undo transactions that have not already​ been saved to the database.

**Syntax:**

* 1. ROLLBACK;

**Example:**

* 1. DELETE FROM CUSTOMERS
  2. WHERE AGE = 25​ ;​
  3. ROLLBACK;

1. **SAVEPOINT:** It is used to roll the transaction back to a certain point without​ rolling back the entire transaction.

**Syntax:**

* 1. SAVEPOINT SAVEPOINT\_NAME;

### 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

● SELECT

**a. SELECT:**​ This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

**Syntax:**

1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;

**For example:**

1. SELECT emp\_name
2. FROM employee
3. WHERE age > ​20​;

**Difference between Open source Software and Commercial Software :**

**S.No. OPEN SOURCE SOFTWARE COMMERCIAL SOFTWARE**

Commercial software is computer

Open source software is computer software where only the person,

software developed either by an

team, or organization that created

individual, group or an organization

1. it can modify it and also they have

to meet certain requirements and it

. exclusive right over the software.

is available openly for the general

Anyone who needs to use it has

public for any modifications based to pay for it with a valid and

on its developing body’s interest. authorized license.

1. The cost of open source software . is free.

|  |  |  |
| --- | --- | --- |
| 03  . | Open source provides limited technical support. | Commercial software provides guaranteed technical support. |
| 04  . | Open source software is available under free licensing. | Commercial software is available under high licensing cost. |
| 05  . | In open source software users need to rely on community support. | In commercial software users get dedicated support from the vendor. |
| 06  . | In open source software installation and updates are administered by the user. | In open source software installation and updates are administered by the software vendor. |

The cost of commercial software varies from moderate to expensive.

Limited hands on training and

1. online resources are available for

. open source software application training.

On site and Online training are available from the commercial software vendor side for software application training.

1. Here in open source software . users can customize.

In this rapid community response

09

helps in fixing the bugs and

. malfunctions.

1. In open source software the . source code is public.

The source of funds of open

1. source software mainly depends

. on donations and support.

Firefox, OpenOffice, Zimbra, VLC

1. media player, Thunderbird are

. some examples of open source software.

**1. MySQL :**

But in commercial software mainly vendors offer customization.

In this mainly the vendor is responsible for fixing the malfunctions.

In commercial software the source code is protected.

The source of commercial software depends on its software sale / product licensing.

Windows Operating System, MS

Office, SAP, Oracle, Adobe Photoshop are some examples of commercial software.

MySQL is an open-source relational database management system (RDBMS) based on Structured Query Language (SQL). It is developed and managed by oracle corporation and initially released on 23 May, 1995. It is widely being used in many small and large scale industrial applications and capable of handling a large volume of data.

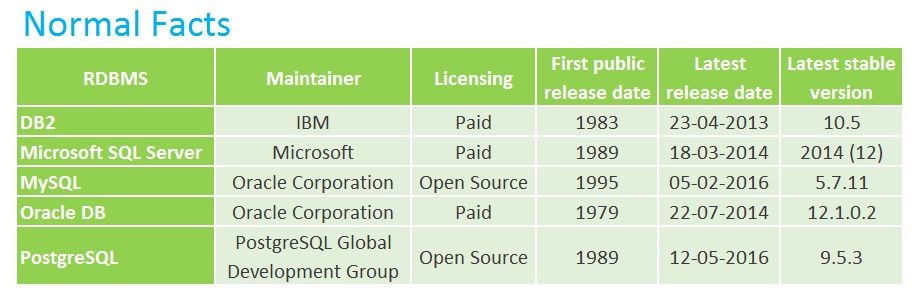
1. **IBM Db2 :**

IBM Db2 is a family of data management products, including database servers, developed by IBM. It is a Relational Database Management System (RDBMS) which supports object-oriented features and non-relational structure with XML. Db2 is designed to store, analyze and retrieve the data efficiently. It was initially released in 1983 and is written in C, C++, Java and Assembly language.

1. **Oracle :**

Oracle is a relational database management system (RDBMS). It was developed by Oracle Corporation in 1980. It is the first database designed for grid computing that provides the most flexible and cost-effective way to manage information and application. It runs on major platforms like Windows, Unix, Linux, and macOS. It is a relational database in which data is accessed by user through an application or query language called SQL.

1. **SQL Server:** is owned and developed by Microsoft Corporation. The​ primary function of SQL Server is the storage and access of data as it is required by other applications, whether they are running on other computers that are connected to a network, or the computer on which the server is stored.



<https://www.geeksforgeeks.org/difference-between-oracle-and-mysql/>

<https://www.geeksforgeeks.org/difference-between-mysql-and-ms-sql-server/>

<https://www.geeksforgeeks.org/difference-between-mysql-and-ibm-db2/>

**Relational database design (RDD):**

Relational database design (RDD) models information and data into a set of tables with rows and columns. Each row of a relation/table represents a record, and each column represents an attribute of data. The Structured Query Language (SQL) is used to manipulate relational databases. The design of a relational database is composed of four stages, where the data are modeled into a set of related tables. The stages are:

* + Define relations/attributes
  + Define primary keys
  + Define relationships
  + Normalization

## Domain

A *domain*​ ​ is the original sets of atomic values used to model data. By *atomic value*​, we mean that each value in the domain is indivisible as far as the relational model is concerned. For example:

* The domain of Marital Status has a set of possibilities: Married, Single, Divorced.
* The domain of Shift has the set of all possible days: {Mon, Tue, Wed…}.
* The domain of Salary is the set of all floating-point numbers greater than 0 and less than 200,000.
* The domain of First Name is the set of character strings that represents names of people.

In summary, a domain is a set of acceptable values that a column is allowed to contain. This is based on various properties and the data type for the column. We will discuss data types in another chapter.

# Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

1. X → Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

**For example:**

Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

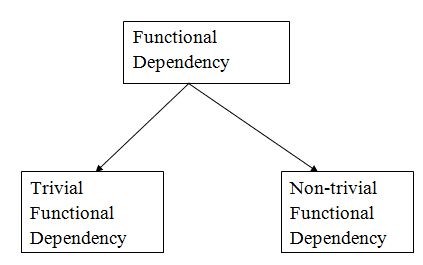
Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of the employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

1. Emp\_Id → Emp\_Name

We can say that Emp\_Name is functionally dependent on Emp\_Id.

## Types of Functional dependency



### 1. Trivial functional dependency

* A → B has trivial functional dependency if B is a subset of A.
* The following dependencies are also trivial like: A → A, B → B

**Example:**

1. Consider a table with two columns Employee\_Id and Employee\_Name.
2. {Employee\_id, Employee\_Name} → Employee\_Id is a trivial functional dependency as
3. Employee\_Id is a subset of {Employee\_Id, Employee\_Name}.
4. Also, Employee\_Id → Employee\_Id and Employee\_Name → Employee\_Name are trivial dependencies too.

### 2. Non-trivial functional dependency

* A → B has a non-trivial functional dependency if B is not a subset of A.
* When A intersection B is NULL, then A → B is called as complete non-trivial.

**Example:**

1. ID → Name,
2. Name → DOB

**Inference Rule (IR):**

* Armstrong's axioms are the basic inference rule.
* Armstrong's axioms are used to conclude functional dependencies on a relational database.
* The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.
* Using the inference rule, we can derive additional functional dependency from

the initial set.

The Functional dependency has 6 types of inference rule:

### 1. Reflexive Rule (IR1​ )​

In the reflexive rule, if Y is a subset of X, then X determines Y.

1. If X ⊇ Y then X → Y

**Example:**

1. X = {a, b, c, d, e}
2. Y = {a, b, c}

### 2. Augmentation Rule (IR2​ )​

The augmentation is also called a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

1. If X → Y then XZ → YZ

**Example:**

1. For R(ABCD), ​**if**​ A → B then AC → BC

### 3. Transitive Rule (IR3​ )​

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

1. If X → Y and Y → Z then X → Z

### 4. Union Rule (IR4​ )​

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

1. If X → Y and X → Z then X → YZ

**Proof:**

1. X → Y (given)
2. X → Z (given)
3. X → XY (using IR​2 on 1 by augmentation with X. Where XX = X)​
4. XY → YZ (using IR​2 on 2 by augmentation with Y)​
5. X → YZ (using IR​3 on 3 and 4)​

### 5. Decomposition Rule (IR5​ )​

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

1. If X → YZ then X → Y and X → Z

**Proof:**

1. X → YZ (given)
2. YZ → Y (using IR​1 Rule)​
3. X → Y (using IR​3 on 1 and 2)​

### 6. Pseudo transitive Rule (IR6​ )​

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

1. If X → Y and YZ → W then XZ → W

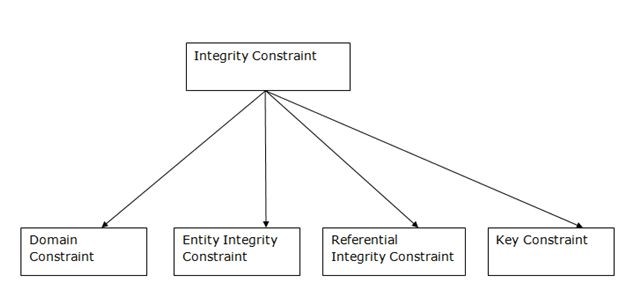
**Proof:**

1. X → Y (given)
2. WY → Z (given)
3. WX → WY (using IR​2 on 1 by augmenting with W)​
4. WX → Z (using IR​3 on 3 and 2)​

# Integrity Constraints

* Integrity constraints are a set of rules. It is used to maintain the quality of information.
* Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
* Thus, integrity constraint is used to guard against accidental damage to the database.

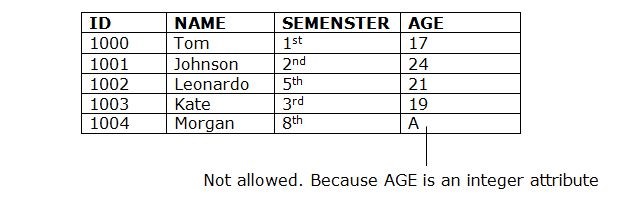
## Types of Integrity Constraint



### 1. Domain constraints

* Domain constraints can be defined as the definition of a valid set of values for an attribute.
* The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

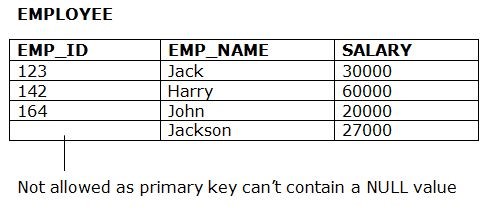
**Example:**



### 2. Entity integrity constraints

* The entity integrity constraint states that primary key value can't be null.
* This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
* A table can contain a null value other than the primary key field.

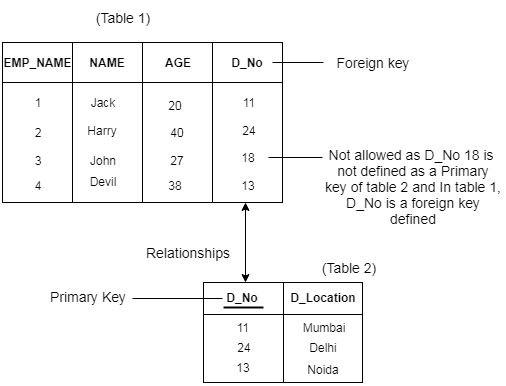
**Example:**



### 3. Referential Integrity Constraints

* A referential integrity constraint is specified between two tables.
* In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

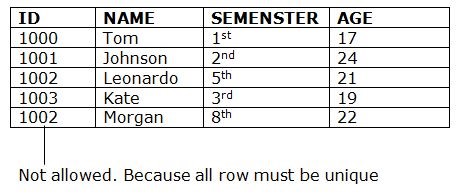
**Example:**



### 4. Key constraints

* Keys are the entity set that is used to identify an entity within its entity set uniquely.
* An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**

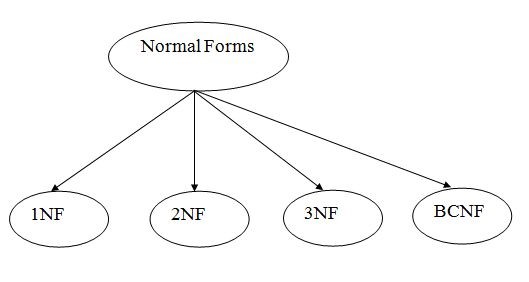


# Normalization

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
* Normalization divides the larger table into the smaller table and links them using relationship.
* The normal form is used to reduce redundancy from the database table.

## Types of Normal Forms

There are the four types of normal forms:



|  |  |
| --- | --- |
| **Norma**  **l Form** | **Description** |
| 1NF | A relation is in 1NF if it contains an atomic value. |
| 2NF | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. |
| 3NF | A relation will be in 3NF if it is in 2NF and no transition dependency exists. |
| 4NF | A relation will be in 4NF if it is in Boyce Codd normal form and has no multivalued dependency. |
| 5NF | A relation is in 5NF if it is in 4NF and does not contain any join dependency and joining should be lossless. |

### First Normal Form (1NF)

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only a single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute​ EMP\_PHONE.

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385,  9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389,  8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

### Second Normal Form (2NF)

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects​ they teach. In a school, a teacher can teach more than one subject.

**TEACHER table**

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

**TEACHER\_DETAIL table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **TEACHER\_AGE** |
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

### Third Normal Form (3NF)

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds at least one of the following conditions for every non-trivial functional dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:**

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **EMP\_ID** | | **EMP\_NAME** | | **EMP\_ZIP** | | **EMP\_STATE** | | **EMP\_CITY** | |
| 222 | | Harry | | 201010 | | UP | | Noida | |
| 333 | | Stephan | | 02228 | | US | | Boston | |
| 444 | | Lan | | 60007 | | US | | Chicago | |
| 555 | | Katharine | | 06389 | | UK | | Norwich | |
| 666 | | John | | 462007 | | MP | | Bhopal | |

**Super key in the table above:**

{EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME,

EMP\_ZIP}....so on

**Candidate key:** {EMP\_ID}​

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are​ non-prime.

Here, EMP\_STATE & EMP\_CITY depend on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively depend on super key(EMP\_ID). It violates the rule of third normal form. That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key. **EMPLOYEE table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_ZIP** |
| 222 | Harry | 201010 |
| 333 | Stephan | 02228 |
| 444 | Lan | 60007 |
| 555 | Katharine | 06389 |
| 666 | John | 462007 |

3.

**EMPLOYEE\_ZIP table:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **EMP\_ZIP** | |  | | **EMP\_STATE** | | **EMP\_CITY** |
| 201010 | | UP | | Noida | | |
| 02228 | | US | | Boston | | |
| 60007 | | US | | Chicago | | |
| 06389 | | UK | | Norwich | | |
| 462007 | | MP | | Bhopal | | |

### Boyce Codd normal form (BCNF)

* BCNF is the advanced version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than​ one department.

**EMPLOYEE table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate key: {EMP-ID, EMP-DEPT}**

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

**EMP\_COUNTRY table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate keys:**

**For the first table:** EMP\_ID​

**For the second table:** EMP\_DEPT​

**For the third table:** {EMP\_ID, EMP\_DEPT}​

Now, this is in BCNF because the left side part of both the functional dependencies is a key.

### Fourth normal form (4NF)

* A relation will be in 4NF if it is in Boyce Codd normal form and has no multivalued dependency.
* For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multivalued dependency.

**Example**

**STUDENT**

|  |  |  |
| --- | --- | --- |
| **STU\_ID** | **COURSE** | **HOBBY** |
| 21 | Computer | Dancing |
| 21 | Math | Singing |
| 34 | Chemistry | Dancing |
| 74 | Biology | Cricket |
| 59 | Physics | Hockey |

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU\_ID, ​**21**​ contains two courses,

**Computer**​ and ​**Math**​ and two hobbies, ​**Dancing**​ and ​**Singing**​. So there is a

Multi-valued dependency on STU\_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

**STUDENT\_COURSE**

|  |  |
| --- | --- |
| **STU\_ID** | **COURSE** |
| 21 | Computer |
| 21 | Math |
| 34 | Chemistry |
| 74 | Biology |
| 59 | Physics |

**STUDENT\_HOBBY**

|  |  |
| --- | --- |
| **STU\_ID** | **HOBBY** |
| 21 | Dancing |
| 21 | Singing |
| 34 | Dancing |
| 74 | Cricket |
| 59 | Hockey |

### Fifth normal form (5NF)

* A relation is in 5NF if it is in 4NF and does not contain any join dependency and joining should be lossless.
* 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
* 5NF is also known as Project-join normal form (PJ/NF).

Example

|  |  |  |
| --- | --- | --- |
| **SUBJECT** | **LECTURER** | **SEMESTER** |
| Computer | Anshika | Semester 1 |
| Computer | John | Semester 1 |
| Math | John | Semester 1 |
| Math | Akash | Semester 2 |
| Chemistry | Praveen | Semester 1 |

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

**P1**

|  |  |  |  |
| --- | --- | --- | --- |
| **SEMESTER** | | **SUBJECT** | |
| Semester 1 | | Computer | |
| Semester 1 | | Math | |
| Semester 1 | | Chemistry | |
| Semester 2 | | Math | |

**P2**

|  |  |
| --- | --- |
| **SUBJECT** | **LECTURER** |
| Computer | Anshika |
| Computer | John |
| Math | John |
| Math | Akash |
| Chemistry | Praveen |

**P3**

|  |  |
| --- | --- |
| **SEMESTER** | **LECTURER** |
| Semester 1 | Anshika |
| Semester 1 | John |
| Semester 1 | John |
| Semester 2 | Akash |
| Semester 1 | Praveen |

**Dependency Preserving:**

* It is an important constraint of the database.
* In the dependency preservation, at least one decomposed table must satisfy every dependency.
* If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.
* For example, suppose there is a relation R (A, B, C, D) with functional dependency set (A->BC). The relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of relation R1(ABC).

**Lossless and Lossy Decomposition in DBMS**

Decomposition in DBMS removes redundancy, anomalies and inconsistencies from a database by dividing the table into multiple tables.

The following are the types −

**Lossless Decomposition:**

Decomposition is lossless if it is feasible to reconstruct relation R from decomposed tables using Joins. This is the preferred choice. The information will not lose from the relation when decomposed. The join would result in the same original relation.

Let us see an example −

#### <EmpInfo>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** | **Dept\_ID** | **Dept\_Name** |
| E001 | Jacob | 29 | Alabama | Dpt1 | Operations |
| E002 | Henry | 32 | Alabama | Dpt2 | HR |
| E003 | Tom | 22 | Texas | Dpt3 | Finance |

Decompose the above table into two tables:

#### <EmpDetails>

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** |
| E001 | Jacob | 29 | Alabama |
| E002 | Henry | 32 | Alabama |
| E003 | Tom | 22 | Texas |

#### <DeptDetails>

|  |  |  |
| --- | --- | --- |
| **Dept\_ID** | **Emp\_ID** | **Dept\_Name** |
| Dpt1 | E001 | Operations |
| Dpt2 | E002 | HR |
| Dpt3 | E003 | Finance |

Now, Natural Join is applied on the above two tables −

The result will be −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** | **Dept\_ID** | **Dept\_Name** |
| E001 | Jacob | 29 | Alabama | Dpt1 | Operation s |
| E002 | Henry | 32 | Alabama | Dpt2 | HR |
| E003 | Tom | 22 | Texas | Dpt3 | Finance |

Therefore, the above relation had lossless decomposition i.e. no loss of information.

## Lossy Decomposition

As the name suggests, when a relation is decomposed into two or more relational schemas, the loss of information is unavoidable when the original relation is retrieved.

Let us see an example −

### <EmpInfo>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** | **Dept\_ID** | **Dept\_Name** |
| E001 | Jacob | 29 | Alabama | Dpt1 | Operation s |
| E002 | Henry | 32 | Alabama | Dpt2 | HR |
| E003 | Tom | 22 | Texas | Dpt3 | Finance |

Decompose the above table into two tables −

### <EmpDetails>

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp\_ID** | **Emp\_Name** | **Emp\_Age** | **Emp\_Location** |
| E001 | Jacob | 29 | Alabama |
| E002 | Henry | 32 | Alabama |
| E003 | Tom | 22 | Texas |

### <DeptDetails>

|  |  |
| --- | --- |
| **Dept\_ID** | **Dept\_Name** |
| Dpt1 | Operations |
| Dpt2 | HR |
| Dpt3 | Finance |

Now, you won’t be able to join the above tables, since **Emp\_ID**​ isn’t​ part of the **DeptDetails** relation.​

Therefore, the above relation has lossy decomposition.

# Query Processing in DBMS

Query Processing is the activity performed in extracting data from the database. In query processing, it takes various steps for fetching the data from the database. The steps involved are:

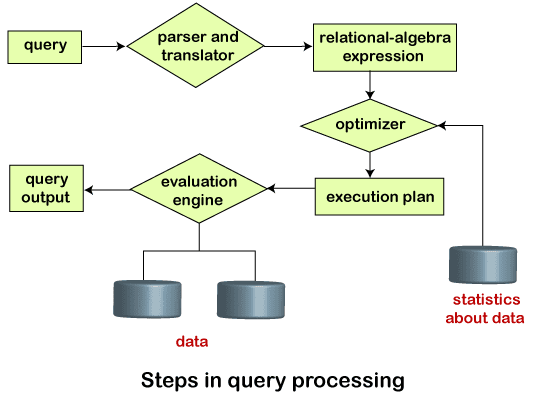
1. Parsing and translation
2. Optimization
3. Evaluation

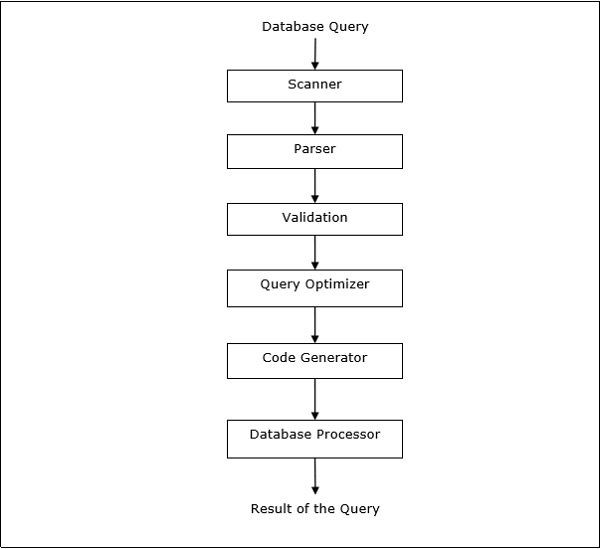
The query processing works in the following way:

## Parsing and Translation

As query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL. It gets translated into expressions that can be further used at the physical level of the file system. After this, the actual evaluation of the queries and a variety of query -optimizing transformations takes place. Thus before processing a query, a computer system needs to translate the query into a human-readable and understandable language. Consequently, SQL or Structured Query Language is the best suitable choice for humans. But, it is not perfectly suitable for the internal representation of the query to the system. Relational algebra is well suited for the internal representation of a query. The translation process in query processing is similar to the parser of a query. When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value. The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in the query.

Thus, we can understand the working of a query processing in the below-described diagram:





Suppose a user executes a query. As we have learned that there are various methods of extracting the data from the database. In SQL, a user wants to fetch the records of the employees whose salary is greater than or equal to 10000. For doing this, the following query is undertaken:

**select emp\_name from Employee where salary>10000;**

Thus, to make the system understand the user query, it needs to be translated in the form of relational algebra. We can bring this query in the relational algebra form as:

* **σsalary>10000**​  **(π**​ **salary**​  **(Employee))**​
* **πsalary**​  **(σ**​ **salary>10000**​  **(Employee))**​

After translating the given query, we can execute each relational algebra operation by using different algorithms. So, in this way, query processing begins working.

## Evaluation

For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.

## Query Evaluation Plan

* In order to fully evaluate a query, the system needs to construct a query evaluation plan.
* The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.
* Such relational algebra with annotations is referred to as **Evaluation**​  **Primitives**. The evaluation primitives carry the instructions needed for the​ evaluation of the operation.
* Thus, a query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as **the**​  **query execution plan**.​
* A **query execution engine**​ is responsible for generating the output of the​ given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

## Optimization

* The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
* Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task is performed by the database system and is known as Query Optimization.
* For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.

Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.

# Query Equivalence Rules

The equivalence rule says that expressions of two forms are the same or equivalent because both expressions produce the same outputs on any legal database instance. It means that we can possibly replace the expression of the first form with that of the second form and replace the expression of the second form with an expression of the first form. Thus, the optimizer of the query-evaluation plan uses such an equivalence rule or method for transforming expressions into the logically equivalent one.

The optimizer uses various equivalence rules on relational-algebra expressions for transforming the relational expressions. For describing each rule, we will use the following symbols:

**θ, θ**​**1, θ**​ ​**2 …**​  : Used for denoting the predicates.​

**L**​**1, L**​ ​**2, L**​ ​**3 …**​  : Used for denoting the list of attributes.​ **E, E**​**1, E**​ ​**2 ….**​  : Represents the relational-algebra expressions.​

Let's discuss a number of equivalence rules:

**Rule 1:** Cascade of σ​

This rule states the deconstruction of the conjunctive selection operations into a sequence of individual selections. Such a transformation is known as a **cascade**​ **of σ**.​ ***σ***​***θ1***~~ᴧ~~ ***θ 2***​ ***(E) = σ***​***θ1***​ ***(σ***​***θ2***​ ***(E))***

**Rule 2:** Commutative Rule​

1. This rule states that selections operations are commutative.

***σ***​***θ1***​ ***(σ***​***θ2***​ ***(E)) = σ*** ​***θ2***​ ***(σ***​***θ1***​ ***(E))***

1. Theta Join (θ) is commutative.

***E***​***1***​⋈​***θ***​ ***E*** ​***2***​ ***= E*** ​***2***​⋈​***θ***​ ***E*** ​***1***​ ***(θ is in subscript with the join symbol)***

However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.

However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.

**Rule 3:** Cascade of ∏​

This rule states that we only need the final operations in the sequence of the projection operations, and other operations are omitted. Such a transformation is referred to as a **cascade of ∏**​ .​

∏L1 (∏L2 (. . . (∏Ln (E)) . . . )) = ∏L1 (E)

**Rule 4:** We can combine the selections with Cartesian products as well as theta joins

**Rule 4:** We can combine the selections with Cartesian products as well as theta joins

1. ***σ***​***θ***​ ***(E***​***1***​ ***x E***​***2***​***) = E***​***1θ***​⋈ ***E***​***2***
2. ***σ***​***θ1***​ ***(E***​***1***​⋈​***θ2***​ ***E***​***2***​***) = E***​***1***​⋈​***θ1***~~ᴧ~~***θ2***​ ***E***​***2***

**Rule 5:** Associative Rule​

1. This rule states that natural join operations are associative.

***(E1*** ⋈ ***E2)*** ⋈ ***E3 = E1*** ⋈ ***(E2*** ⋈ ***E3)***

1. Theta joins are associative for the following expression:

***(E***​***1***​⋈​***θ1***​ ***E***​***2***​***)*** ⋈​***θ2***~~ᴧ~~***θ3***​ ***E***​***3***​ ***= E***​***1***​⋈​***θ1***~~ᴧ~~***θ3***​ ***(E***​***2***​⋈​***θ2***​ ***E***​***3***​***)***

In the theta associativity, θ2 involves the attributes from E2 and E3 only. There may be chances of empty conditions, and thereby it concludes that Cartesian Product is also associative.

|  |
| --- |
| Note: The equivalence rules of associativity and commutatively of join operations are |
| essential for join reordering in query optimization. |

**Rule 6:** Distribution of the Selection operation over the Theta join.​

Under two following conditions, the selection operation gets distributed over the theta-join operation:

1. When all attributes in the selection condition θ​0 include only attributes of one of the expressions which are being joined.

***σ***​***θ0***​ ***(E***​***1***​⋈​***θ***​ ***E***​***2***​***) = (σ***​***θ0***​ ***(E***​***1***​***))*** ⋈​***θ***​ ***E***​***2***

1. When the selection condition θ​1 involves the attributes of E​1 only, and θ​2 includes the attributes of E​2 only.​

***σ***​***θ1 θ2***​ ***(E***​***1***​⋈​***θ***​ ***E***​***2***​***) = (σ***​***θ1***​ ***(E***​***1***​***))*** ⋈​***θ*** ​***((σ***​***θ2***​ ***(E***​***2***​***))***

**Rule 7:** Distribution of the projection operation over the theta join.​

Under two following conditions, the selection operation gets distributed over the theta-join operation:

1. Assume that the join condition θ includes only in L​1 υ L​2 attributes of E​1 and E​2 Then, we get the following expression:

***∏***​***L1υL2***​ ***(E***​***1***​⋈​***θ***​ ***E***​***2***​***) = (∏***​***L1***​ ***(E***​***1***​***))*** ⋈​***θ***​ ***(∏***​***L2***​ ***(E***​***2***​***))***

1. Assume a join as E​1 ⋈ E​2.​ Both expressions E​1 and E​2 have sets of attributes as L​1 and L​2.​ Assume two attributes L​3 and L​4 where L​3 be attributes of the expression E​1,​ involved in the θ join condition but not in L​1 υ L​2 Similarly, an L​4 be attributes of the expression E​2involved​ only in the θ join condition and not in L​1 υ L​2attributes.​ Thus, we get the following expression:

***∏***​***L1υL2***​ ***(E***​***1***​⋈​***θ***​ ***E***​***2***​***) = ∏***​***L1υL2***​ ***((∏***​***L1υL3***​ ***(E***​***1***​***))*** ⋈​***θ***​ ***((∏***​***L2υL4***​ ***(E***​***2***​***)))***

**Rule 8:** The union and intersection set operations are commutative.​

***E***​***1***​ ***υ E***​***2***​ ***= E***​***2***​ ***υ E***​***1***

***E***​***1***​ ***E***​***2***​ ***= E***​***2***​ ***E***​***1***

However, set difference operations are not commutative.

**Rule 9:** The union and intersection set operations are associative.​

***(E***​***1***​ ***υ E***​***2***​***) υ E***​***3***​ ***= E***​***1***​ ***υ (E***​***2***​ ***υ E***​***3***​***)***

***(E***​***1***​ ***E***​***2***​***) E***​***3***​ ***= E***​***1***​ ***(E***​***2***​ ***E***​***3***​***)***

**Rule 10:** Distribution of selection operation on the intersection, union, and set difference operations.

The below expression shows the distribution performed over the set difference operation.

***σ***​***p***​ ***(E***​***1***​ ***− E***​***2***​***) = σ***​***p***​***(E***​***1***​***) − σ***​***p***​***(E***​***2***​***)***

We can similarly distribute the selection operation on υ and by replacing with -. Further, we get:

***σ***​***p***​ ***(E***​***1***​ ***− E***​***2***​***) = σ***​***p***​***(E***​***1***​***) −E***​***2***

**Rule 11**:​ Distribution of the projection operation over the union operation.

This rule states that we can distribute the projection operation on the union operation for the given expressions.

***∏***​***L***​ ***(E***​***1***​ ***υ E***​***2***​***) = (∏***​***L*** ​***(E***​***1***​***)) υ (∏***​***L*** ​***(E***​***2***​***))***

Apart from these discussed equivalence rules, there are various other equivalence rules also.

# SQL JOIN

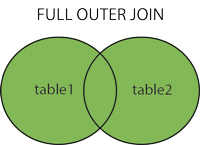
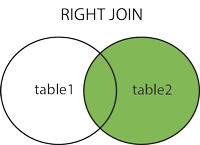
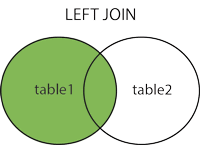
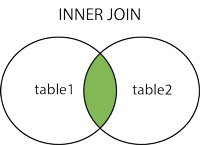
As the name shows, JOIN means to combine something. In case of SQL, JOIN means "to combine two or more tables".

In SQL, JOIN clause is used to combine the records from two or more tables in a database.

## Types of SQL JOIN

1. INNER JOIN 2. LEFT JOIN

1. RIGHT JOIN
2. FULL JOIN



Sample Table

**EMPLOYEE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **CITY** | **SALARY** | **AGE** |
| 1 | Angelina | Chicago | 200000 | 30 |
| 2 | Robert | Austin | 300000 | 26 |
| 3 | Christian | Denver | 100000 | 42 |
| 4 | Kristen | Washington | 500000 | 29 |
| 5 | Russell | Los angeles | 200000 | 36 |
| 6 | Marry | Canada | 600000 | 48 |

**PROJECT**

|  |  |  |
| --- | --- | --- |
| **PROJECT\_NO** | **EMP\_ID** | **DEPARTMENT** |
| 101 | 1 | Testing |
| 102 | 2 | Development |
| 103 | 3 | Designing |
| 104 | 4 | Development |

### 1. INNER JOIN

In SQL, INNER JOIN selects records that have matching values in both tables as long as the condition is satisfied. It returns the combination of all rows from both the tables where the condition satisfies.

**Syntax**

1. SELECT table1.column1, table1.column2, table2.column1,....
2. FROM table1
3. INNER JOIN table2
4. ON table1.matching\_column​ = ​ table2​ .matching\_column;​

**Query**

1. SELECT EMPLOYEE.EMP\_NAME, PROJECT.DEPARTMENT
2. FROM EMPLOYEE
3. INNER JOIN PROJECT
4. ON PROJECT.EMP\_ID​ = ​ EMPLOYEE​ .EMP\_ID;​

**Output**

|  |  |
| --- | --- |
| **EMP\_NAME** | **DEPARTMENT** |
| Angelina | Testing |
| Robert | Development |
| Christian | Designing |
| Kristen | Development |

### 2. LEFT JOIN

The SQL left join returns all the values from left table and the matching values from the right table. If there is no matching join value, it will return NULL.

**Syntax**

1. SELECT table1.column1, table1.column2, table2.column1,....
2. FROM table1
3. LEFT JOIN table2
4. ON table1.matching\_column​ = ​ table2​ .matching\_column;​

**Query**

1. SELECT EMPLOYEE.EMP\_NAME, PROJECT.DEPARTMENT
2. FROM EMPLOYEE
3. LEFT JOIN PROJECT
4. ON PROJECT.EMP\_ID​ = ​ EMPLOYEE​ .EMP\_ID;​

**Output**

|  |  |
| --- | --- |
| **EMP\_NAME** | **DEPARTMENT** |
| Angelina | Testing |
| Robert | Development |
| Christian | Designing |
| Kristen | Development |
| Russell | NULL |
| Marry | NULL |

### 3. RIGHT JOIN

In SQL, RIGHT JOIN returns all the values from the values from the rows of right table and the matched values from the left table. If there is no matching in both tables, it will return NULL.

**Syntax**

1. SELECT table1.column1, table1.column2, table2.column1,....
2. FROM table1
3. RIGHT JOIN table2
4. ON table1.matching\_column​ = ​ table2​ .matching\_column;​

**Query**

1. SELECT EMPLOYEE.EMP\_NAME, PROJECT.DEPARTMENT
2. FROM EMPLOYEE
3. RIGHT JOIN PROJECT
4. ON PROJECT.EMP\_ID​ = ​ EMPLOYEE​ .EMP\_ID;​

**Output**

|  |  |
| --- | --- |
| **EMP\_NAME** | **DEPARTMENT** |
| Angelina | Testing |
| Robert | Development |
| Christian | Designing |
| Kristen | Development |

### 4. FULL JOIN

In SQL, FULL JOIN is the result of a combination of both left and right outer join. Join tables have all the records from both tables. It puts NULL on the place of matches not found.

**Syntax**

1. SELECT table1.column1, table1.column2, table2.column1,....
2. FROM table1
3. FULL JOIN table2
4. ON ​table1.matching\_column​ = ​table2​.matching\_column;

**Query**

1. SELECT EMPLOYEE.EMP\_NAME, PROJECT.DEPARTMENT
2. FROM EMPLOYEE
3. FULL JOIN PROJECT
4. ON ​PROJECT.EMP\_ID​ = ​EMPLOYEE​.EMP\_ID;

**Output**

|  |  |
| --- | --- |
| **EMP\_NAME** | **DEPARTMENT** |
| Angelina | Testing |
| Robert | Development |
| Christian | Designing |
| Kristen | Development |
| Russell | NULL |
| Marry | NULL |

### UNIT - 2 COMPLETED