DATA:

- Data is Raw, unorganized facts that need to be processed.
- Data comprises facts, figures, observations, numbers, characters, symbols, images, etc.
- Data is a known facts about any entity.
- Set of Data that is processed in meaningful way according to the given requirement is called Information.
- *For Example:- 100, Amazon, R No., Name*
- Collection of Interrelated Data called Record.

Ex.	R No	Name	Subject	Marks
	01	John	POD	18

DATABASE:

Database is a collection of interrelated data i.e. *Records*.

- Database Systems are designed to manage large amount of Data.
- The Database system must ensure the safety of the Data.
- Database System applications:

1. ***Banking:-*** *For customer info, accounts, loans, and banking transactions.*
2. ***Airlines/Trains :-*** *For Reservations and schedule Information.*
3. ***Universities:-*** *For Student information, Course registrations, Results.*
4. ***Telecommunication:-*** *For keeping records of calls made. For Billing*

Database Purpose

- Provides secure and survivable medium for storage & retrieval of Data.
- Data shared among several users & is persistent.
- Provides mechanism to create, access and manipulate data without compromising security and integrity of Data.
- Redundancy can be reduced.
- Inconsistency can be avoided.

CHARACTERISTICS OF DATABASE:

- **Real-world entity:-** We should be able to store all kinds of data that exist in this real world. Since we need to work with all kinds of data and requirements, the database should be strong enough to store all kinds of data that are present around us.
- **Relation-based tables:-** We should be able to relate the entities/tables in the database by means of relation. Ex. An employee works for a department. This implies that an Employee is related to a particular department.
- **Isolation of data and application:-** Data and applications should be isolated. Because the database is a system that gives the platform to store the data, and the data is the one that allows the database to work. Hence there should be a clear differentiation between them.
- **Less redundancy**:- There should not be any duplication of data in the database. Data should be stored in such a way that it should not be repeated in multiple tables. If repeated, it would be an unnecessary waste of DB space, and maintaining such data becomes chaos.

- **Consistency:-** Consistency is a state where data cannot be written that would violate the database's own rules for valid data. If a certain transaction occurs that attempts to introduce inconsistent data, the entire transaction is rolled back and an error returned to the user.
- **Query Language**:- DBMS has a strong query language. Once the database is designed, this helps the user to retrieve and manipulate the data. If a particular user wants to see any specific data, he can apply as many filtering conditions that he wants and pull the data that he needs.
- **Multiuser and Concurrent Access:-** Multiple users should be able to access the same database, without affecting the other user. i.e.; if teachers want to update a student's marks in the Results table at the same time, then they should be allowed to update the marks for their subjects, without modifying other subject marks.
- **Multiple views**:- It supports multiple view to the user, depending on his role. In a school database, Students will able to see only their reports and their access would be read-only. At the same time, teachers will have access to all the students with modification rights. But the database is the same. Hence a single database provides different views to different users.
- **Security**:- The database should also provide security, i.e.; when there are multiple users are accessing the database, each user will have their own levels of rights to see the database. Some of them will be allowed to see the whole database, and some will have only partial rights. For example, an instructor who is teaching Physics will have access to see and update marks of his subject. He will not have access to other subjects. But the HOD will have full access to all the subjects.
- **ACID Properties:-** The database should also support the ACID property. i.e.; while performing any transactions like insert, update and delete, the database makes sure that the real purpose of the data is not lost. For example, if a student's address is updated, then it

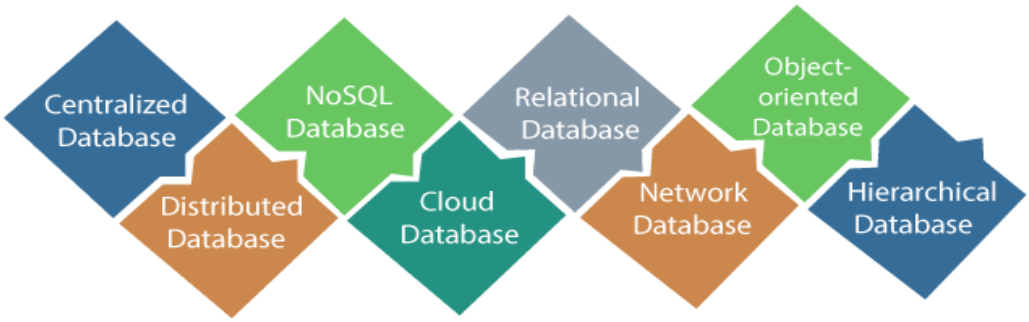
should make sure that there is no duplicate data is created nor there is any data mismatch for that student.

FILE SYSTEM V/S DATABASE APPROACH :

File based system	**Database system**
1. The data and program are inter- dependent.	1. The data and program are independent of each other.
2. File-based system caused data redundancy. The data may be duplicated in different files	2. Database system control data redundancy. The data appeared only once in the system.
3. File –based system caused data inconsistency. The data in different files may be different that cause data inconsistency.	3. In database system data always consistent. Because data appeared only once.
4. The data cannot be shared because data is distributed in different files.	4. In database data is easily shared because data is stored at one place.
5. In file based system data is widely spread. Due to this reason file based system provides poor security.	5. It provides many methods to maintain data security in the database.
6. File based system does not provide consistency constrains.	6. Database system provides a different consistency constrains to maintain data integrity in the system.
7. File based system is less complex system.	7. Database system is very complex system.

8. To generate different report to take a crucial decision is very difficult in file based system.	8. The report can be generated very easily in required format in database system.
9. File based system takes much space in the system, and memory is wasted in this approach.	9. Database approach store data more efficiently it takes less space in the system and memory is not wasted.
10. File based system does not provide concurrency facility.	10. Database system provides concurrency facility.
11. File based system does not provide data atomicity functionality.	11. Database system provides data atomicity functionality.

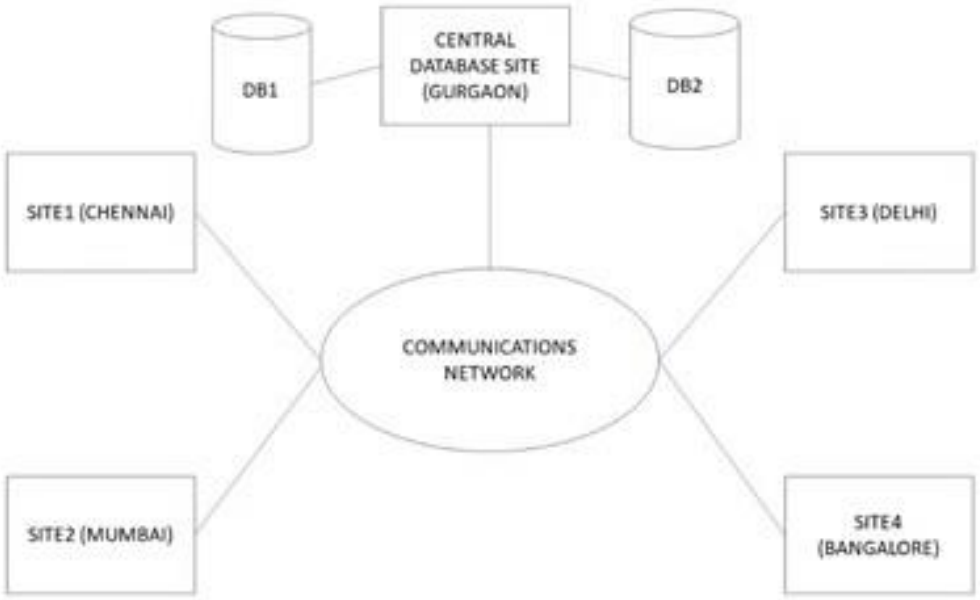
TYPES OF DATABASE :



1. Centralized Database

The information(data) is stored at a centralized location and the users from different locations can access this data. This type of database contains application procedures that help the users to access the data even from a remote location.

Various kinds of authentication procedures are applied for the verification and validation of end users, likewise, a registration number is provided by the application procedures which keeps a track and record of data usage. The local area office handles this thing



2.DISTRIBUTED DATABASE

Just opposite of the centralized database concept, the distributed database has contributions from the common database as well as the information captured by local computers also. The data is not at one place and is distributed at various sites of an organization. These sites are connected to each other with the help of communication links which helps them to access the distributed data easily.

You can imagine a distributed database as a one in which various portions of a database are stored in multiple different locations(physical) along with the application procedures which are replicated and distributed among various points in a network.

There are two kinds of distributed database, viz. homogenous and heterogeneous. The databases which have same underlying hardware and run over the same operating systems and application procedures are known as homogeneous DDB, for eg. All physical locations in a DDB. Whereas, the operating systems, underlying hardware as well as application procedures can be different at various sites of a DDB which is known as heterogeneous DDB.

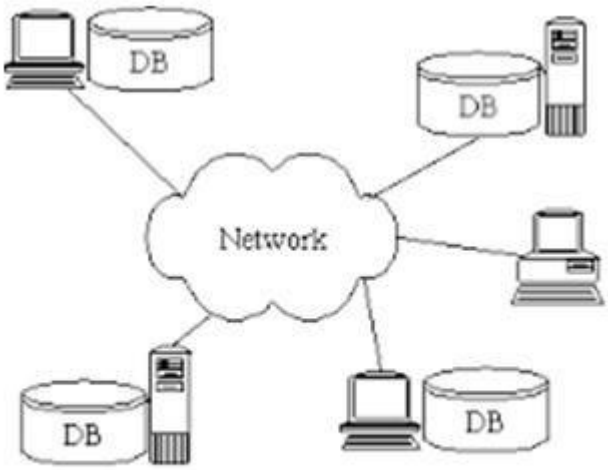
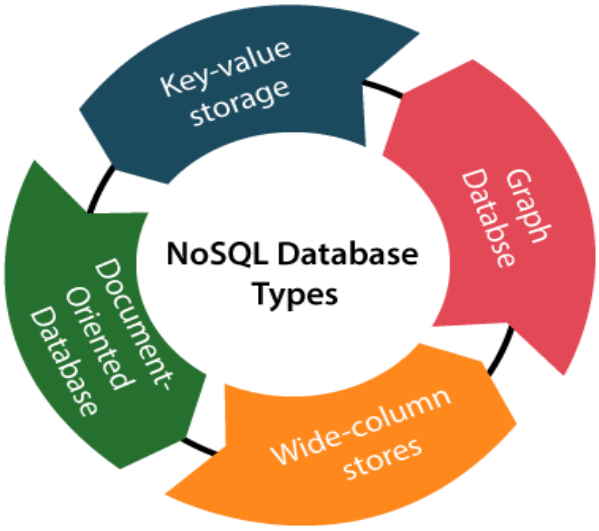


Figure. Distributed database System

3. NOSQL DATABASE

Non-SQL/Not Only SQL is a type of database that is used for storing a wide range of data sets. It is not a relational database as it stores data not only in tabular form but in several different ways. It came into existence when the demand for building modern applications increased. Thus, NoSQL presented a wide variety of database technologies in response to the demands. We can further divide a NoSQL database into the following four types:



5. CLOUD DATABASE

A type of database where data is stored in a virtual environment and executes over the cloud computing platform. It provides users with various cloud computing services (SaaS, PaaS, IaaS, etc.) for accessing the database. There are numerous cloud platforms, but the best options are:

Amazon Web Services(AWS)

Microsoft Azure

Kamatera

PhonixNAP

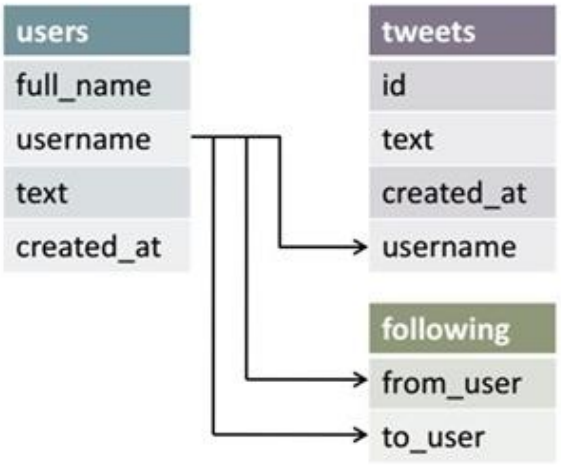
ScienceSoft

Google Cloud SQL, etc.

5. RELATIONAL DATABASE

These databases are categorized by a set of tables where data gets fit into a pre-defined category. The table consists of rows and columns where the column has an entry for data for a specific category and rows contains instance for that data defined according to the category. The Structured Query Language (SQL) is the standard user and application program interface for a relational database.

There are various simple operations that can be applied over the table which makes these databases easier to extend, join two databases with a common relation and modify all existing applications.



6. NETWORK DATABASE

It is the database that typically follows the network data model.

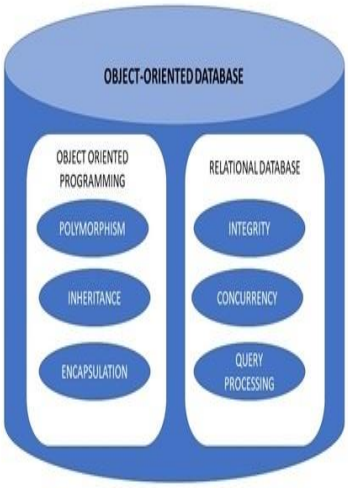
Here, the representation of data is in the form of nodes connected via links between them.

Unlike the hierarchical database, it allows each record to have multiple children and parent nodes to form a generalized graph structure.

7. OBJECT ORIENTED DATABASE

An object-oriented database is a collection of object-oriented programming and relational database. There are various items which are created using object-oriented programming languages like C++, Java which can be stored in relational databases, but object-oriented databases are well-suited for those items.

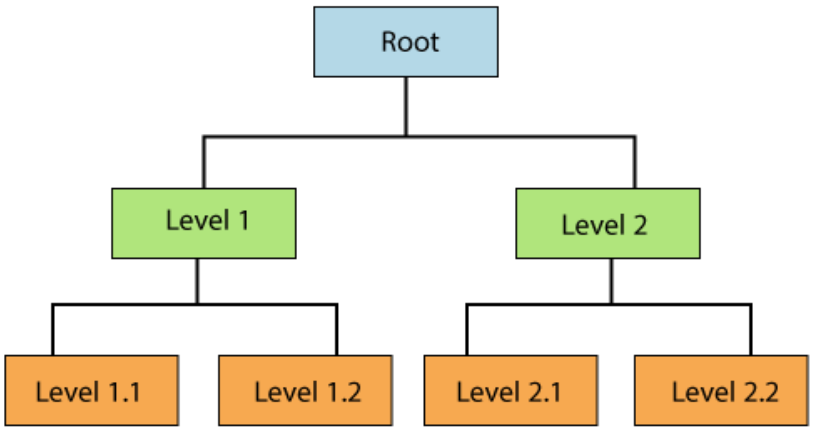
An object-oriented database is organized around objects rather than actions, and data rather than logic. For example, a multimedia record in a relational database can be a definable data object, as opposed to an alphanumeric value.



8. HIERARCHICAL DATABASE

It is the type of database that stores data in the form of parent-children relationship nodes. Here, it organizes data in a tree-like structure.

Data get stored in the form of records that are connected via links. Each child record in the tree will contain only one parent. On the other hand, each parent record can have multiple child records.



Hierarchical Database

ADVANTAGES OF DATABASE

- Reduced data redundancy
- Reduced updating errors and increased consistency
- Greater data integrity and independence from applications programs
- Improved data access to users through use of host and query languages
- Improved data security
- Reduced data entry, storage, and retrieval costs
- Facilitated development of new applications program

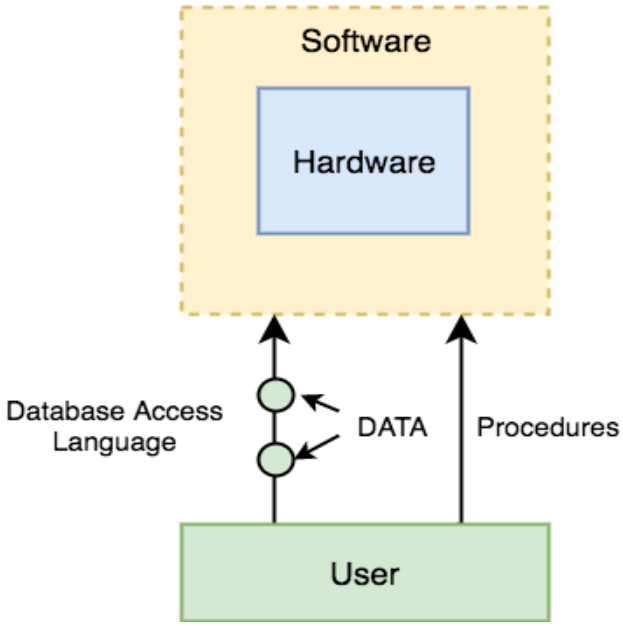
DISADVANTAGES OF DATABASE

- Database systems are complex, difficult, and time-consuming to design
- Substantial hardware and software start-up costs
- Damage to database affects virtually all applications programs
- Extensive conversion costs in moving form a file-based system to a database system
- Initial training required for all programmers and users

COMPONENTS OF DATABASE

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

- **Hardware**
- **Software**
- **Data**
- **Procedures**
- **Database Access Language**
- **Users**



HARDWARE

- When we say Hardware, we mean computer, hard disks, I/O channels for data, and any other physical component involved before any data is successfully stored into the memory.
- When we run Oracle or MySQL on our personal computer, then our computer's Hard Disk, our Keyboard using which we type in all the commands, our computer's RAM, ROM all become a part of the DBMS hardware.

SOFTWARE

- The main component of a Database management system is the software. It is the set of programs which is used to manage the database and to control the overall computerized database.
- The DBMS software provides an easy-to-use interface to store, retrieve, and update data in the database.
- This software component is capable of understanding the Database Access Language and converts it into actual database commands to execute or run them on the database.

DATA

- It is the most important component of the database management system.
- The main task of DBMS is to process the data. Here, databases are defined, constructed, and then data is stored, retrieved, and updated to and from the databases.
- The database contains both the metadata (description about data or data about data) and the actual (or operational) data.

PROCEDURES

- Procedures refer to general rules and instructions that help to design the database and to use a database management system.
- Procedures are used to setup and install a new database management system (DBMS), to login and logout of DBMS software, to manage DBMS or application programs, to take backup of the database, and to change the structure of the database, etc.

DATABASE ACCESS LANGUAGE

- Database Access Language is a simple language designed to write commands to access, insert, update and delete data stored in any database.
- A user can write commands in the Database Access Language and submit it to the DBMS for execution, which is then translated and executed by the DBMS.
- User can create new databases, tables, insert data, fetch stored data, update data and delete the data using the access language.

USERS

The users are the people who control and manage the databases and perform different types of operations on the databases in the database management system.

There are three types of user who play different roles in DBMS:

1. Application Programmers: The users who write the application programs in programming languages (such as Java, C++, or Visual Basic) to interact with databases are called Application Programmer.

2. Database Administrators (DBA): A person who manages the overall DBMS is called a database administrator or simply DBA.

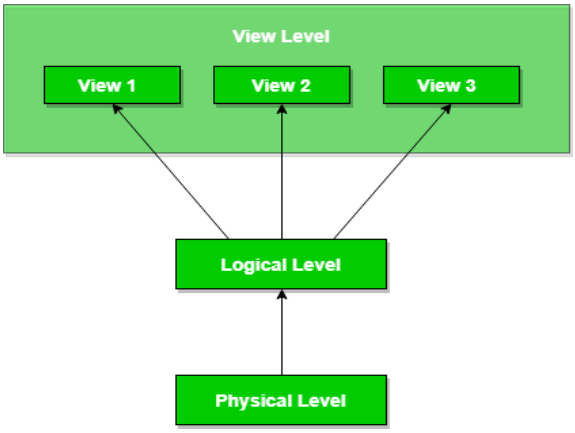
3. End-Users: The end-users are those who interact with the database management system to perform different operations by using the different database commands such as insert, update, retrieve, and delete on the data, etc.

Data Abstraction

Database systems comprise of complex data-structures. In order to make the system efficient in terms of retrieval of data, and reduce complexity in terms of usability of users, developers use abstraction i.e. hide irrelevant details from the users. This approach simplifies database design.

There are mainly **3** levels of data abstraction:

1. Physical
2. Logical
3. View



Physical: This is the lowest level of data abstraction. It tells us how the data is actually stored in memory. The access methods like sequential or random access and file organization methods like B+ trees, hashing used for the same. Usability, size of memory, and the number of times the records are factors which we need to know while designing the database.
Suppose we need to store the details of an employee. Blocks of storage and the amount of memory used for these purposes is kept hidden from the user.

Logical: This level comprises of the information that is actually stored in the database in the form of tables. It also stores the relationship among the data entities in relatively simple structures. At this level, the information available to the user at the view level is unknown.
We can store the various attributes of an employee and relationships, e.g. with the manager can also be stored.

View: This is the highest level of abstraction. Only a part of the actual database is viewed by the users. This level exists to ease the accessibility of the database by an individual user. Users view data in the form of rows and columns. Tables and relations are used to store data. Multiple views of the same database may exist. Users can just view the data and interact with the database, storage and implementation details are hidden from them.

DATABASE LANGUAGES

- A DBMS has appropriate languages and interfaces to express database queries and updates.
- Database languages can be used to read, store and update the data in the database.

Types of Database Language

1. **Data Definition Language (DDL) :-**

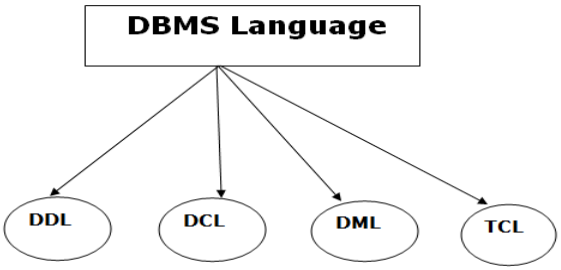
- It is used to define database structure or pattern.
- It is used to create schema, tables, indexes, constraints, etc. in the database.
- Using the DDL statements, you can create the skeleton of the database.
- Data definition language is used to store the information of metadata like the number of tables and schemas, their names, indexes, columns in each table, constraints, etc.

Here are some tasks that come under DDL:

- **Create:** It is used to create objects in the database.
- **Alter:** It is used to alter the structure of the database.
- **Drop:** It is used to delete objects from the database.
- **Truncate:** It is used to remove all records from a table.
- **Rename:** It is used to rename an object.

- **Comment:** It is used to comment on the data dictionary.

These commands are used to update the database schema that's why they come under Data definition language.



2. Data Manipulation Language

- **DML** stands for **D**ata **M**anipulation **L**anguage. It is used for accessing and manipulating data in a database. It handles user requests.

Here are some tasks that come under DML:

Select: It is used to retrieve data from a database.

Insert: It is used to insert data into a table.

Update: It is used to update existing data within a table.

Delete: It is used to delete all records from a table.

Merge: It performs UPSERT operation, i.e., insert or update operations.

Call: It is used to call a structured query language or a Java subprogram.

Explain Plan: It has the parameter of explaining data.

Lock Table: It controls concurrency.

3. Data Control Language

- **DCL** stands for **D**ata **C**ontrol **L**anguage. It is used to retrieve the stored or saved data.
- The DCL execution is transactional. It also has rollback parameters.

Here are some tasks that come under DCL:

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

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

There are the following operations which have the authorization of Revoke:

CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

4. Transaction Control Language

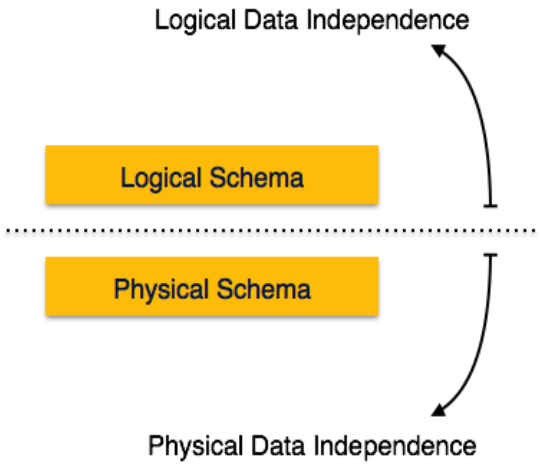
- TCL is used to run the changes made by the DML statement. TCL can be grouped into a logical transaction.

Here are some tasks that come under TCL:

- **Commit:** It is used to save the transaction on the database.
- **Rollback:** It is used to restore the database to original since the last Commit.

DATA INDEPENDENCE

- A database system normally contains a lot of data in addition to users' data.
- For example, it stores data about data, known as metadata, to locate and retrieve data easily.
- It is rather difficult to modify or update a set of metadata once it is stored in the database.
- But as a DBMS expands, it needs to change over time to satisfy the requirements of the users.
- If the entire data is dependent, it would become a tedious and highly complex job.



Logical Data Independence

- Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

- Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

Physical Data Independence

- All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.
- For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

DATA INTEGRITY

- The term *data integrity* refers to the accuracy and consistency of data.
- When creating databases, attention needs to be given to data integrity and how to maintain it. A good database will enforce data integrity whenever possible.
- For example, a user could accidentally try to enter a phone number into a date field. If the system enforces data integrity, it will prevent the user from making these mistakes.

  Maintaining data integrity means making sure the data remains intact and unchanged throughout its entire life cycle. This includes the capture of the data, storage, updates, transfers, backups, etc. Every time data is processed there's a risk that it could get corrupted (whether accidentally or maliciously

4 Types of Data Integrity

1. Entity Integrity

Entity integrity defines each row to be unique within its table. No two rows can be the same.

To achieve this, a primary key can be defined. The primary key field contains a unique identifier – no two rows can contain the same unique identifier.

2. Referential Integrity

Referential integrity is concerned with relationships. When two or more tables have a relationship, we have to ensure that the foreign key value matches the primary key value at all times. We don't want to have a situation where a foreign key value has no matching primary key value in the primary table. This would result in an orphaned record.

3. Domain Integrity

Domain integrity concerns the validity of entries for a given column. Selecting the appropriate data type for a column is the first step in maintaining domain integrity. Other steps could include, setting

up appropriate constraints and rules to define the data format and/or restricting the range of possible values.

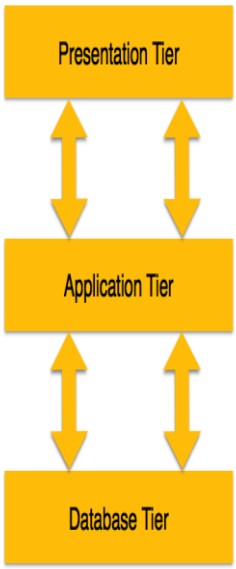
THREE LEVEL ARCHITECTURE FOR DB

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.

Database (Data) Tier – At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.

Application (Middle) Tier – At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.

User (Presentation) Tier – End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.



12 CODD'S RULE:

Here is brief note on E.F Codd's Twelve rules:

Rule 0 − Foundation rule

Any relational database management system that is propounded to be RDBMS or advocated to be a RDBMS should be able to manage the stored data in its entirety through its relational capabilities.

Rule 1 − Rule of Information

Relational Databases should store the data in the form of relations. Tables are relations in Relational Database Management Systems. Be it any user defined data or meta-data, it is important to store the value as an entity in the table cells.

Rule 2 − Rule of Guaranteed Access

The use of pointers to access data logically is strictly forbidden. Every data entity which is atomic in nature should be accessed logically by using a right combination of the name of table, primary key represented by a specific row value and column name represented by attribute value.

Rule 3 − Rule of Systematic Null Value Support

Null values are completely supported in relational databases. They should be uniformly considered as 'missing information'. Null values are independent of any data type. They should not be mistaken for blanks or zeroes or empty strings. Null values can also be interpreted as 'inapplicable data' or 'unknown information.'

Rule 4 − Rule of Active and online relational Catalog

In the Database Management Systems lexicon, 'metadata' is the data about the database or the data about the data. The active online catalog that stores the metadata is called 'Data dictionary'. The so called data dictionary is accessible only by authored users who have the required privileges and the query languages used for accessing the database should be used for accessing the data of data dictionary.

Rule 5 − Rule of Comprehensive Data Sub-language

A single robust language should be able to define integrity constraints, views, data manipulations, transactions and authorizations. If the database allows access to the aforementioned ones, it is violating this rule.

Rule 6 − Rule of Updating Views

Views should reflect the updates of their respective base tables and vice versa. A view is a logical table which shows restricted data. Views generally make the data readable but not modifiable. Views help in data abstraction.

Rule 7 − Rule of Set level insertion, update and deletion

A single operation should be sufficient to retrieve, insert, update and delete the data.

Rule 8 − Rule of Physical Data Independence

Batch and end user operations are logically separated from physical storage and respective access methods.

Rule 9 − Rule of Logical Data Independence

Batch and end users can change the database schema without having to recreate it or recreate the applications built upon it.

Rule 10 − Rule of Integrity Independence

Integrity constraints should be available and stored as metadata in data dictionary and not in the application programs.

Rule 11 − Rule of Distribution Independence

The Data Manipulation Language of the relational system should not be concerned about the physical data storage and no alterations should be required if the physical data is centralized or distributed.

Rule 12 − Rule of Non Subversion

Any row should obey the security and integrity constraints imposed. No special privileges are applicable.

Oracle String Function

The Oracle String functions manipulate the character strings more effectively.

The following table indicates each of the functions with a brief description:

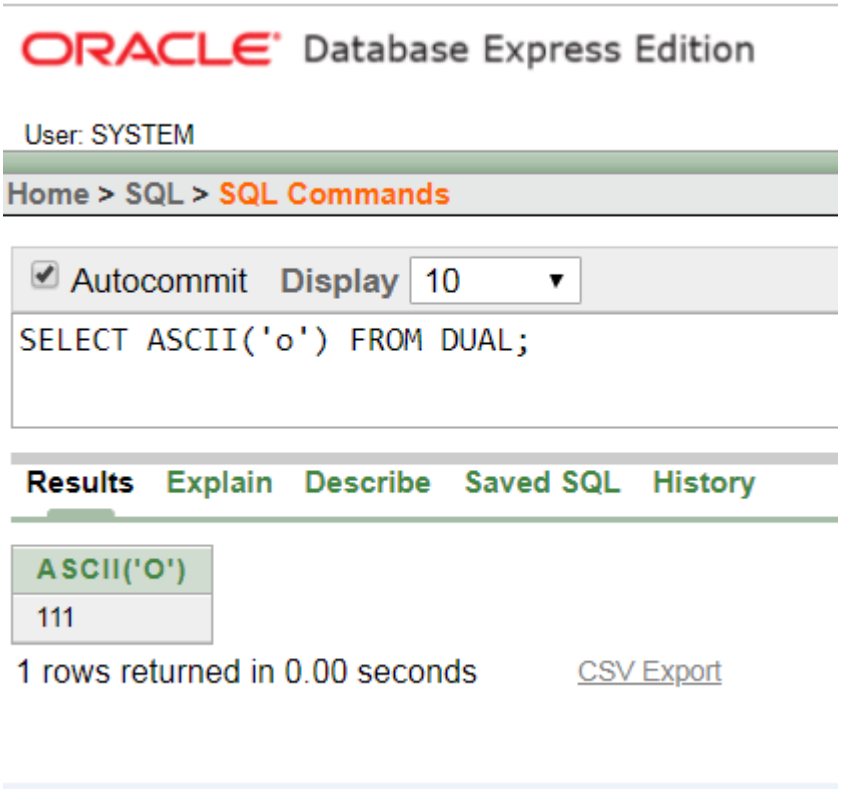
Functions	Description
ASCII()	The ASCII() function returns the numeric value of given character.
ASCIISTR()	The ASCIISTR() function returns the numeric value of given character.
CHR()	The Oracle CHR() function returns all the characters of the given ASCII code.
COMPOSE()	The Oracle COMPOSE() function returns the Unicode string.
CONCAT()	The Oracle CONCAT() function returns a string by concatenating all the arguments.
CONCAT WITH ||	The Oracle CONCAT WITH || function returns a string by concatenating all the arguments.
CONVERT()	The Oracle CONVERT() function is used to convert one string set to another.
DECOMPOSE()	The DECOMPOSE() function returns a Unicode string from the given string.
DUMP()	The DUMP() function returns the string datatype code, length in bytes and some internal representation of the given string.
INSTR4()	The Oracle INSTR4() function returns the location of the substring using UCS4 code points from the given string.
INSTRB()	The Oracle INSTRB() function returns the location of substring using bytes instead of characters from the given string.
INSTRC()	The INSTRC() function returns the location of the substring using Unicode complete characters from the given string.
LENGTH2()	The LENGTH2() function returns the size using UCS2 code points of the given string.
LENGTH4()	The Oracle LENGTH4() function returns the size using UCS4 code points of the given string.
LENGTHB()	The LENGTHB() function returns the size using bytes instead of characters from the given string.
LENGTHC()	The LENGTHC() function returns the size using Unicode complete characters from the given string.
LENGTH()	The Oracle LENGTH() function returns the size of the given string.
LOWER()	The Oracle LOWER() function returns the given string into the lower case.
LPAD()	The LPAD() returns the left padded to the given length.
LTRIM()	The LTRIM() function returns the string by removing given characters from the left side.
NCHR()	The NCHR() function returns the character which is based on the number_code in the national character set.

REGEXP_INSTR()	The Oracle REGEXP_INSTR() function is used for pattern matching in a string.
REGEXP_REPLACE()	The REGEXP_REPLACE() function is used for pattern matching and replacing the sequence of characters.
REGEXP_SUBSTR()	The REGEXP_SUBSTR()function is used for pattern matching in substring from a string.
REPLACE()	The Oracle REPLACE() function is used to replace the sequence of character with another character in the given string.
RPAD()	The RPAD() function returns the right-padded to the given length.
RTRIM()	The Oracle RTRIM() function returns the string by removing given characters from the right side.
SOUNDEX()	The SOUNDEX() function returns a Soundex string which sounds almost same and have identical soundex strings.
SUBSTR()	The Oracle SUBSTR() function is used to extract the substring from the given string.
TRANSLATE()	The TRANSLATE() function is used to replace the character from the given character. This function replaces one character only.
TRIM()	The Oracle TRIM() function is used to remove the specified character from the head of the string or tail of the string.
UPPER()	The Upper() function is used to convert the given string into uppercase.
VSIZE()	The Oracle VSIZE() function returns the number of bytes of the given expression.

Example 1

1. SELECT ASCII('o') FROM DUAL

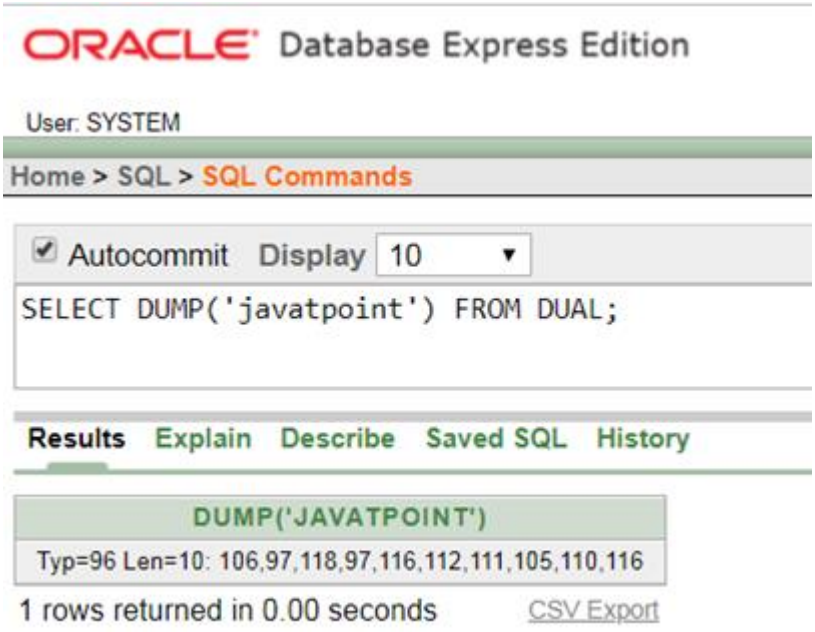
Output:



Example 2

1. SELECT DUMP('javatpoint') FROM DUAL;

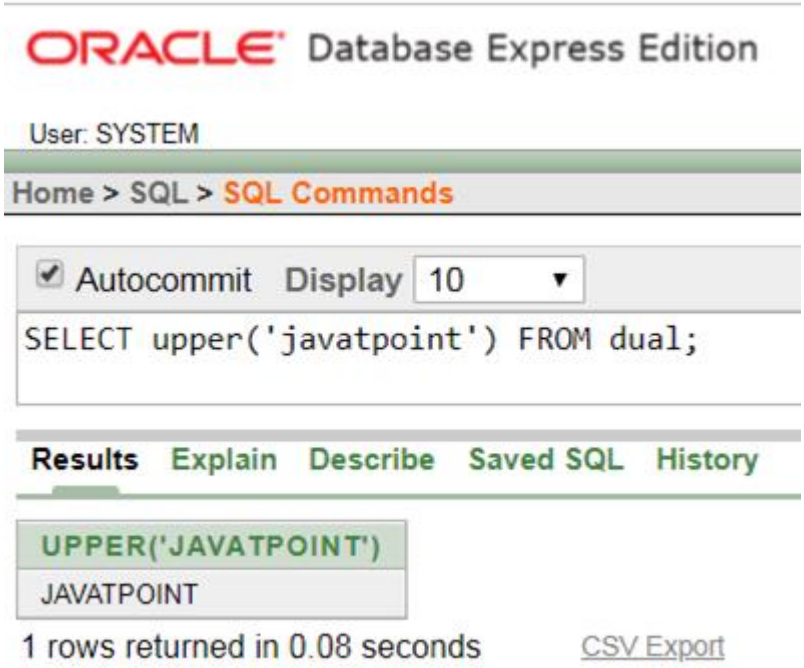
Output:



Example 3

1. SELECT upper('javatpoint') FROM dual;

Output:


ORACLE Database Express Edition
User: SYSTEM
Home > SQL > SQL Commands
Autocommit Display 10
SELECT upper('javatpoint') FROM dual;
Results Explain Describe Saved SQL History
UPPER('JAVATPOINT')
JAVATPOINT
1 rows returned in 0.08 seconds
CSV Export

Oracle Date Functions

Function	Example	Result	Description
ADD_MONTHS	`ADD_MONTHS( DATE '2016-02-29', 1 )`	`31-MAR-16`	Add a number of months (n) to a date and return the same day which is n of months away.
CURRENT_DATE	`SELECT CURRENT_DATE FROM dual`	`06-AUG-2017 19:43:44`	Return the current date and time in the session time zone
CURRENT_TIMESTAMP	`SELECT CURRENT_TIMESTAMP FROM dual`	`06-AUG-17 08.26.52.742000000 PM -07:00`	Return the current date and time with time zone in the session time zone
DBTIMEZONE	`SELECT DBTIMEZONE FROM dual;`	`-07:00`	Get the current database time zone
EXTRACT	`EXTRACT(YEAR FROM SYSDATE)`	`2017`	Extract a value of a date time field e.g., YEAR, MONTH, DAY, … from a date time value.
FROM_TZ	`FROM_TZ(TIMESTAMP '2017-08-08 08:09:10', '-09:00')`	`08-AUG-17 08.09.10.000000000 AM -07:00`	Convert a timestamp and a time zone to a `TIMESTAMP WITH TIME ZONE` value
LAST_DAY	`LAST_DAY(DATE '2016-02-01')`	`29-FEB-16`	Gets the last day of the month of a specified date.
LOCALTIMESTAMP	`SELECT LOCALTIMESTAMP FROM dual`	`06-AUG-17 08.26.52.742000000 PM`	Return a `TIMESTAMP` value that represents the current date

			and time in the session time zone.
MONTHS_BETWEEN	MONTHS_BETWEEN(DATE '2017-07-01', DATE '2017-01-01')	6	Return the number of months between two dates.
NEW_TIME	NEW_TIME(TO_DATE('08-07-2017 01:30:45', 'MM-DD-YYYY HH24:MI:SS'), 'AST', 'PST')	06-AUG-2017 21:30:45	Convert a date in one time zone to another
NEXT_DAY	NEXT_DAY(DATE '2000-01-01', 'SUNDAY')	02-JAN-00	Get the first weekday that is later than a specified date.
ROUND	ROUND(DATE '2017-07-16', 'MM')	01-AUG-17	Return a date rounded to a specific unit of measure.
SESSIONTIMEZONE	SELECT SESSIONTIMEZONE FROM dual;	-07:00	Get the session time zone
SYSDATE	SYSDATE	01-AUG-17	Return the current system date and time of the operating system where the Oracle Database resides.
SYSTIMESTAMP	SELECT SYSTIMESTAMP FROM dual;	01-AUG-17 01.33.57.929000000 PM -07:00	Return the system date and time that includes fractional seconds and time zone.
TO_CHAR	TO_CHAR(DATE'2017-01-01', 'DL')	Sunday, January 01, 2017	Convert a DATE or an INTERVAL value to a character string in a specified format.
TO_DATE	TO_DATE('01 Jan 2017', 'DD MON YYYY')	01-JAN-17	Convert a date which is in the character string

			to a `DATE` value.
TRUNC	`TRUNC(DATE '2017-07-16', 'MM')`	`01-JUL-17`	Return a date truncated to a specific unit of measure.
TZ_OFFSET	`TZ_OFFSET( 'Europe/London' )`	`+01:00`	Get time zone offset of a time zone name from UTC

SQL Operators

Every database administrator and user uses SQL queries for manipulating and accessing the data of database tables and views.

The manipulation and retrieving of the data are performed with the help of reserved words and characters, which are used to perform arithmetic operations, logical operations, comparison operations, compound operations, etc.

What is SQL Operator?

The SQL reserved words and characters are called operators, which are used with a WHERE clause in a SQL query. In SQL, an operator can either be a unary or binary operator. The unary operator uses only one operand for performing the unary operation, whereas the binary operator uses two operands for performing the binary operation.

Syntax of Unary SQL Operator

1. Operator SQL_Operand

Syntax of Unary SQL Operator

1. Operand1 SQL_Operator Operand2

Note: SQL operators are used for filtering the table's data by a specific condition in the SQL statement.

What is the Precedence of SQL Operator?

The precedence of SQL operators is the sequence in which the SQL evaluates the different operators in the same expression. Structured Query Language evaluates those operators first, which have high precedence.

In the following table, the operators at the top have high precedence, and the operators that appear at the bottom have low precedence.

SQL Operator Symbols	Operators
**	Exponentiation operator
+, -	Identity operator, Negation operator
*, /	Multiplication operator, Division operator
+, -, ||	Addition (plus) operator, subtraction (minus) operator, String Concatenation operator
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison Operators
NOT	Logical negation operator
&& or AND	Conjunction operator
OR	Inclusion operator

For Example,

```
1. UPDATE employee
2. SET salary = 20 - 3 * 5 WHERE Emp_Id = 5;
```

In the above SQL example, salary is assigned 5, not 85, because the * (Multiplication)

Operator has higher precedence than the - (subtraction) operator, so it first gets multiplied with 3*5 and then subtracts from 20.

Types of Operator

SQL operators are categorized in the following categories:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators
4. SQL Set Operators
5. SQL Bit-wise Operators
6. SQL Unary Operators

Let's discuss each operator with their types.

SQL Arithmetic Operators

The **Arithmetic Operators** perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.

Following are the various arithmetic operators performed on the SQL data:

1. SQL Addition Operator (+)
2. SQL Subtraction Operator (-)
3. SQL Multiplication Operator (+)
4. SQL Division Operator (-)
5. SQL Modulus Operator (+)

SQL Addition Operator (+)

The **Addition Operator** in SQL performs the addition on the numerical data of the database table. In SQL, we can easily add the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also add the numbers to the existing numbers of the specific column.

Syntax of SQL Addition Operator:

```
1. SELECT operand1 + operand2;
```

Let's understand the below example which explains how to execute Addition Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.**

Emp Id	Emp Name	Emp Salary	Emp Monthlybonus
101	Tushar	25000	4000
102	Anuj	30000	200

- Suppose, we want to add **20,000** to the salary of each employee specified in the table. Then, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary + 20000 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL addition operation on the single column of the given table.

- Suppose, we want to add the Salary and monthly bonus columns of the above table, then we have to write the following query in SQL:

1. SELECT Emp_Salary + Emp_Monthlybonus as Emp_Total_Salary FROM Employee_details;

In this query, we have added two columns with each other of the above table.

SQL Subtraction Operator (-)

The Subtraction Operator in SQL performs the subtraction on the numerical data of the database table. In SQL, we can easily subtract the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also subtract the number from the existing number of the specific table column.

Syntax of SQL Subtraction Operator:

1. SELECT operand1 - operand2;

Let's understand the below example which explains how to execute Subtraction Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.**

Emp Id	Emp Name	Emp Salary	Penalty
201	Abhay	25000	200
202	Sumit	30000	500

- Suppose we want to subtract 5,000 from the salary of each employee given in the **Employee_details** table. Then, we have to write the following query in the SQL:

1. SELECT Emp_Salary - 5000 as Emp_New_Salary FROM Employee_details;

In this query, we have performed the SQL subtraction operation on the single column of the given table.

- If we want to subtract the penalty from the salary of each employee, then we have to write the following query in SQL:

1. SELECT Emp_Salary - Penalty as Emp_Total_Salary FROM Employee_details;

SQL Multiplication Operator (*)

The Multiplication Operator in SQL performs the Multiplication on the numerical data of the database table. In SQL, we can easily multiply the numerical values of two columns of the same table by specifying both the column names as the first and second operand.

Syntax of SQL Multiplication Operator:

1. SELECT operand1 * operand2;

Let's understand the below example which explains how to execute Multiplication Operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.**

Emp Id	Emp Name	Emp Salary	Penalty
201	Abhay	25000	200

202 Sumit 30000 500

- Suppose, we want to double the salary of each employee given in the **Employee_details** table. Then, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary * 2 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL multiplication operation on the single column of the given table.

- If we want to multiply **the Emp_Id** column to **Emp_Salary** column of that employee whose **Emp_Id** is **202,** then we have to write the following query in SQL:

```
1. SELECT Emp_Id * Emp_Salary as Emp_Id * Emp_Salary FROM Employee_details WHERE Em
   p_Id = 202;
```

In this query, we have multiplied the values of two columns by using the WHERE clause.

SQL Division Operator (/)

The Division Operator in SQL divides the operand on the left side by the operand on the right side.

Syntax of SQL Division Operator:

```
1. SELECT operand1 / operand2;
```

In SQL, we can also divide the numerical values of one column by another column of the same table by specifying both column names as the first and second operand.

We can also perform the division operation on the stored numbers in the column of the SQL table.

Let's understand the below example which explains how to execute Division Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	25000
202	Sumit	30000

- Suppose, we want to half the salary of each employee given in the Employee_details table. For this operation, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary / 2 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL division operation on the single column of the given table.

SQL Modulus Operator (%)

The Modulus Operator in SQL provides the remainder when the operand on the left side is divided by the operand on the right side.

Syntax of SQL Modulus Operator:

1. SELECT operand1 % operand2;

Let's understand the below example which explains how to execute Modulus Operator in SQL query:

This example consists of a **Division** table, which has three columns **Number, First_operand, and Second_operand.**

Number	First operand	Second operand
1	56	4
2	32	8
3	89	9
4	18	10
5	10	5

- If we want to get the remainder by dividing the numbers of First_operand column by the numbers of Second_operand column, then we have to write the following query in SQL:

1. SELECT First_operand % Second_operand as Remainder FROM Employee_details;

SQL Comparison Operators

The **Comparison Operators** in SQL compare two different data of SQL table and check whether they are the same, greater, and lesser. The SQL comparison operators are used with the WHERE clause in the SQL queries

Following are the various comparison operators which are performed on the data stored in the SQL database tables:

1. SQL Equal Operator (=)
2. SQL Not Equal Operator (!=)
3. SQL Greater Than Operator (>)
4. SQL Greater Than Equals to Operator (>=)
5. SQL Less Than Operator (<)\
6. SQL Less Than Equals to Operator (<=)

SQL Equal Operator (=)

This operator is highly used in SQL queries. The **Equal Operator** in SQL shows only data that matches the specified value in the query.

This operator returns TRUE records from the database table if the value of both operands specified in the query is matched.

Let's understand the below example which explains how to execute Equal Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	30000
202	Ankit	40000
203	Bheem	30000
204	Ram	29000
205	Sumit	30000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is 30000. Then, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Salary = 30000;
```

In this example, we used the SQL equal operator with WHERE clause for getting the records of those employees whose salary is 30000.

SQL Equal Not Operator (!=)

The **Equal Not Operator** in SQL shows only those data that do not match the query's specified value.

This operator returns those records or rows from the database views and tables if the value of both operands specified in the query is not matched with each other.

Let's understand the below example which explains how to execute Equal Not Operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is not 45000. Then, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Salary != 45000;
```

In this example, we used the SQL equal not operator with WHERE clause for getting the records of those employees whose salary is not 45000.

SQL Greater Than Operator (>)

The **Greater Than Operator** in SQL shows only those data which are greater than the value of the right-hand operand.

Let's understand the below example which explains how to execute Greater ThanOperator (>) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is greater than 202. Then, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id > 202;
```

Here, SQL greater than operator displays the records of those employees from the above table whose Employee Id is greater than 202.

SQL Greater Than Equals to Operator (>=)

The **Greater Than Equals to Operator** in SQL shows those data from the table which are greater than and equal to the value of the right-hand operand.

Let's understand the below example which explains how to execute greater than equals to the operator (>=) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000

204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is greater than and equals to 202. For this, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id >= 202;
```

Here,'**SQL greater than equals to operator**' with WHERE clause displays the rows of those employees from the table whose Employee Id is greater than and equals to 202.

SQL Less Than Operator (<)

The **Less Than Operator** in SQL shows only those data from the database tables which are less than the value of the right-side operand.

This comparison operator checks that the left side operand is lesser than the right side operand. If the condition becomes true, then this operator in SQL displays the data which is less than the value of the right-side operand.

Let's understand the below example which explains how to execute less than operator (<) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is less than 204. For this, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id < 204;
```

Here,**SQL less than operator** with WHERE clause displays the records of those employees from the above table whose Employee Id is less than 204.

SQL Less Than Equals to Operator (<=)

The **Less Than Equals to Operator** in SQL shows those data from the table which are lesser and equal to the value of the right-side operand.

This comparison operator checks that the left side operand is lesser and equal to the right side operand.

Let's understand the below example which explains how to execute less than equals to the operator (<=) in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, and Emp_Salary.**

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose employee id is less and equals **203.** For this, we have to write the following query in the SQL database:

1. SELECT * FROM Employee_details WHERE Emp_Id <= 203;

Here, SQL **less than equals to the operator** with WHERE clause displays the rows of those employees from the table whose Employee Id is less than and equals 202.

SQL Logical Operators

The **Logical Operators** in SQL perform the Boolean operations, which give two results **True and False.** These operators provide **True** value if both operands match the logical condition.

Following are the various logical operators which are performed on the data stored in the SQL database tables:

1. SQL ALL operator
2. SQL AND operator
3. SQL OR operator
4. SQL BETWEEN operator
5. SQL IN operator
6. SQL NOT operator
7. SQL ANY operator
8. SQL LIKE operator

SQL ALL Operator

The ALL operator in SQL compares the specified value to all the values of a column from the sub-query in the SQL database.

This operator is always used with the following statement:

1. SELECT,
2. HAVING, and
3. WHERE.

Syntax of ALL operator:

1. SELECT column_Name1, ..., column_NameN FROM table_Name WHERE column Compariso n_operator ALL (SELECT column FROM tablename2)

Let's understand the below example which explains how to execute ALL logical operators in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Gurgaon
202	Ankit	45000	Delhi
203	Bheem	30000	Jaipur
204	Ram	29000	Mumbai
205	Sumit	40000	Kolkata

- If we want to access the employee id and employee names of those employees from the table whose salaries are greater than the salary of employees who lives in Jaipur city, then we have to type the following query in SQL.

1. SELECT Emp_Id, Emp_Name FROM Employee_details WHERE Emp_Salary > ALL (SELECT Em p_Salary FROM Employee_details WHERE Emp_City = Jaipur)

Here, we used the **SQL ALL operator** with greater than the operator.

SQL AND Operator

The **AND operator** in SQL would show the record from the database table if all the conditions separated by the AND operator evaluated to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of AND operator:

1. SELECT column1, ..., columnN FROM table_Name WHERE condition1 AND condition2 AND condition3 AND ..... AND conditionN;

Let's understand the below example which explains how to execute AND logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to access all the records of those employees from the **Employee_details** table whose salary is 25000 and the city is Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi';

Here,**SQL AND operator** with WHERE clause shows the record of employees whose salary is 25000 and the city is Delhi.

SQL OR Operator

The OR operator in SQL shows the record from the table if any of the conditions separated by the OR operator evaluates to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of OR operator:

1. SELECT column1, ..., columnN FROM table_Name WHERE condition1 OR condition2 OR con dition3 OR ..... OR conditionN;

Let's understand the below example which explains how to execute OR logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- If we want to access all the records of those employees from the **Employee_details** table whose salary is 25000 or the city is Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi';

Here, **SQL OR operator** with WHERE clause shows the record of employees whose salary is 25000 or the city is Delhi.

SQL BETWEEN Operator

The **BETWEEN operator** in SQL shows the record within the range mentioned in the SQL query. This operator operates on the numbers, characters, and date/time operands.

If there is no value in the given range, then this operator shows NULL value.

Syntax of BETWEEN operator:

1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE co lumn_nameBETWEEN value1 and value2;

Let's understand the below example which explains how to execute BETWEEN logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to access all the information of those employees from the **Employee_details** table who is having salaries between 20000 and 40000. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary BETWEEN 30000 AND 45000;

Here, we used the **SQL BETWEEN operator** with the Emp_Salary field.

SQL IN Operator

The **IN operator** in SQL allows database users to specify two or more values in a WHERE clause. This logical operator minimizes the requirement of multiple OR conditions.

This operator makes the query easier to learn and understand. This operator returns those rows whose values match with any value of the given list.

Syntax of IN operator:

1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE co lumn_name IN (list_of_values);

Let's understand the below example which explains how to execute IN logical operator in SQL query:

This example consists of an **Employee_details** table, which has three columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is 202, 204, and 205. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Id IN (202, 204, 205);
```

Here, we used the **SQL IN operator** with the Emp_Id column.

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose **Employee Id** is not equal to 202 and 205. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Id NOT IN (202,205);
2.
```

Here, we used the **SQL NOT IN operator** with the Emp_Id column.

SQL NOT Operator

The **NOT operator** in SQL shows the record from the table if the condition evaluates to false. It is always used with the WHERE clause.

Syntax of NOT operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name WHERE NOT condition;
```

Let's understand the below example which explains how to execute NOT logical operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose Cityis not Delhi. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' ;
```

In this example, we used the **SQL NOT operator** with the Emp_City column.

- Suppose, we want to show all the information of those employees from the **Employee_details** table whose Cityis not Delhi and Chandigarh. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' AND NOT Emp_City = 'Ch
   andigarh';
```

In this example, we used the **SQL NOT operator** with the Emp_City column.

SQL ANY Operator

The **ANY operator** in SQL shows the records when any of the values returned by the sub-query meet the condition.

The ANY logical operator must match at least one record in the inner query and must be preceded by any SQL comparison operator.

Syntax of ANY operator:

1. SELECT column1, column2 ...., columnN FROM table_Name WHERE column_name comparis on_operator ANY (SELECT column_name FROM table_name WHERE condition(s)) ;

SQL LIKE Operator

The **LIKE operator** in SQL shows those records from the table which match with the given pattern specified in the sub-query.

The percentage (%) sign is a wildcard which is used in conjunction with this logical operator.

This operator is used in the WHERE clause with the following three statements:

1. SELECT statement
2. UPDATE statement
3. DELETE statement

Syntax of LIKE operator:

1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE co lumn_name LIKE pattern;

Let's understand the below example which explains how to execute LIKE logical operator in SQL query:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- If we want to show all the information of those employees from the **Employee_details** whose name starts with ''s''. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Name LIKE 's%' ;
```

In this example, we used the SQL LIKE operator with **Emp_Name** column because we want to access the record of those employees whose name starts with s.

- If we want to show all the information of those employees from the **Employee_details**whose name ends with ''y''. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Name LIKE '%y' ;
```

- If we want to show all the information of those employees from the **Employee_details**whose name starts with ''S'' and ends with ''y''. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Name LIKE 'S%y' ;
```

SQL Set Operators

The **Set Operators** in SQL combine a similar type of data from two or more SQL database tables. It mixes the result, which is extracted from two or more SQL queries, into a single result.

Set operators combine more than one select statement in a single query and return a specific result set.

Following are the various set operators which are performed on the similar data stored in the two SQL database tables:

1. SQL Union Operator
2. SQL Union ALL Operator
3. SQL Intersect Operator
4. SQL Minus Operator

SQL Union Operator

The SQL Union Operator combines the result of two or more SELECT statements and provides the single output.

The data type and the number of columns must be the same for each SELECT statement used with the UNION operator. This operator does not show the duplicate records in the output table.

Syntax of UNION Set operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
```

2. UNION
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];

Let's understand the below example which explains how to execute Union operator in Structured Query Language:

In this example, **we used two tables.** Both tables have four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

- Suppose, we want to see the employee name and employee id of each employee from both tables in a single output. For this, we have to write the following query in SQL:

1. SELECT Emp_ID, Emp_Name FROM Employee_details1
2. UNION
3. SELECT Emp_ID, Emp_Name FROM Employee_details2 ;

SQL Union ALL Operator

The SQL Union Operator is the same as the UNION operator, but the only difference is that it also shows the same record.

Syntax of UNION ALL Set operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
2. UNION ALL
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];
```

Let's understand the below example which explains how to execute Union ALL operator in Structured Query Language:

In this example, **we used two tables.** Both tables have four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

- If we want to see the employee name of each employee of both tables in a single output. For this, we have to write the following query in SQL:

```
1. SELECT Emp_Name FROM Employee_details1
2. UNION ALL
3. SELECT Emp_Name FROM Employee_details2 ;
```

SQL Intersect Operator

The SQL Intersect Operator shows the common record from two or more SELECT statements. The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.

Syntax of INTERSECT Set operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
2. INTERSECT
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];
```

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, **we used two tables.** Both tables have four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see a common record of the employee from both the tables in a single output. For this, we have to write the following query in SQL:

```
1. SELECT Emp_Name FROM Employee_details1
2. INTERSECT
3. SELECT Emp_Name FROM Employee_details2 ;
```

SQL Minus Operator

The SQL Minus Operator combines the result of two or more SELECT statements and shows only the results from the first data set.

Syntax of MINUS operator:

```
1. SELECT column1, column2 ...., columnN FROM First_tablename [WHERE conditions]
2. MINUS
3. SELECT column1, column2 ...., columnN FROM Second_tablename [WHERE conditions];
```

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, **we used two tables.** Both tables have four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see the name of employees from the first result set after the combination of both tables. For this, we have to write the following query in SQL:

```
1. SELECT Emp_Name FROM Employee_details1
2. MINUS
3. SELECT Emp_Name FROM Employee_details2 ;
```

SQL Unary Operators

The **Unary Operators** in SQL perform the unary operations on the single data of the SQL table, i.e., these operators operate only on one operand.

These types of operators can be easily operated on the numeric data value of the SQL table.

Following are the various unary operators which are performed on the numeric data stored in the SQL table:

1. SQL Unary Positive Operator
2. SQL Unary Negative Operator
3. SQL Unary Bitwise NOT Operator

SQL Unary Positive Operator

The SQL Positive (+) operator makes the numeric value of the SQL table positive.

Syntax of Unary Positive Operator

1. SELECT +(column1), +(column2) ...., +(columnN) FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute a Positive unary operator on the data of SQL table:

This example consists of an**Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to see the salary of each employee as positive from the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT +Emp_Salary Employee_details ;

SQL Unary Negative Operator

The SQL Negative (-) operator makes the numeric value of the SQL table negative.

Syntax of Unary Negative Operator

1. SELECT -(column_Name1), -(column_Name2) ...., -(column_NameN) FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute Negative unary operator on the data of SQL table:

This example consists of an **Employee_details** table, which has four columns **Emp_Id, Emp_Name, Emp_Salary, and Emp_City.**

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to see the salary of each employee as negative from the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT -Emp_Salary Employee_details ;

- Suppose, we want to see the salary of those employees as negative whose city is Kolkatain the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT -Emp_Salary Employee_details WHERE Emp_City = 'Kolkata';

SQL Bitwise NOT Operator

The SQL Bitwise NOT operator provides the one's complement of the single numeric operand. This operator turns each bit of numeric value. If the bit of any numerical value is 001100, then this operator turns these bits into 110011.

Syntax of Bitwise NOT Operator

1. SELECT ~(column1), ~(column2) ...., ~(columnN) FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute the Bitwise NOT operator on the data of SQL table:

This example consists of a**Student_details** table, which has four columns **Roll_No, Stu_Name, Stu_Marks, and Stu_City.**

Emp Id	Stu Name	Stu Marks	Stu City
101	Sanjay	85	Delhi
102	Ajay	97	Chandigarh

103	Saket	45	Delhi
104	Abhay	68	Delhi
105	Sumit	60	Kolkata

If we want to perform the Bitwise Not operator on the marks column of **Student_details,** we have to write the following query in SQL:

```
1. SELECT ~Stu_Marks Employee_details ;
```

SQL Bitwise Operators

The **Bitwise Operators** in SQL perform the bit operations on the Integer values. To understand the performance of Bitwise operators, you just knew the basics of Boolean algebra.

Following are the two important logical operators which are performed on the data stored in the SQL database tables:

1. Bitwise AND (&)
2. Bitwise OR(|)

Bitwise AND (&)

The Bitwise AND operator performs the logical AND operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise AND Operator

```
1. SELECT column1 & column2 & .... & columnN FROM table_Name [WHERE conditions] ;
```

Let's understand the below example which explains how to execute Bitwise AND operator on the data of SQL table:

This example consists of the following table, which has two columns. Each column holds numerical values.

When we use the Bitwise AND operator in SQL, then SQL converts the values of both columns in binary format, and the AND operation is performed on the converted bits.

After that, SQL converts the resultant bits into user understandable format, i.e., decimal format.

Column1	Column2
1	1
2	5
3	4
4	2
5	3

- Suppose, we want to perform the Bitwise AND operator between both the columns of the above table. For this, we have to write the following query in SQL:

```
1. SELECT Column1 & Column2 From TABLE_AND ;
```

Bitwise OR (|)

The Bitwise OR operator performs the logical OR operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise OR Operator

1. SELECT column1 | column2 | .... | columnN FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute Bitwise OR operator on the data of SQL table:

This example consists of a table that has two columns. Each column holds numerical values.

When we used the Bitwise OR operator in SQL, then SQL converts the values of both columns in binary format, and the OR operation is performed on the binary bits. After that, SQL converts the resultant binary bits into user understandable format, i.e., decimal format.

Column1	Column2
1	1
2	5
3	4
4	2
5	3

- Suppose, we want to perform the Bitwise OR operator between both the columns of the above table. For this, we have to write the following query in SQL:

1. SELECT Column1 | Column2 From TABLE_OR ;

Next TopicSQL CREATE Database

Normalization in DBMS: 1NF, 2NF, 3NF and BCNF in Database

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss about anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

Normalization

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says “each attribute of a table must have atomic (single) values”, the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}
Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:
teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:
emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:
For first table: emp_id
For second table: emp_dept
For third table: {emp_id, emp_dept}

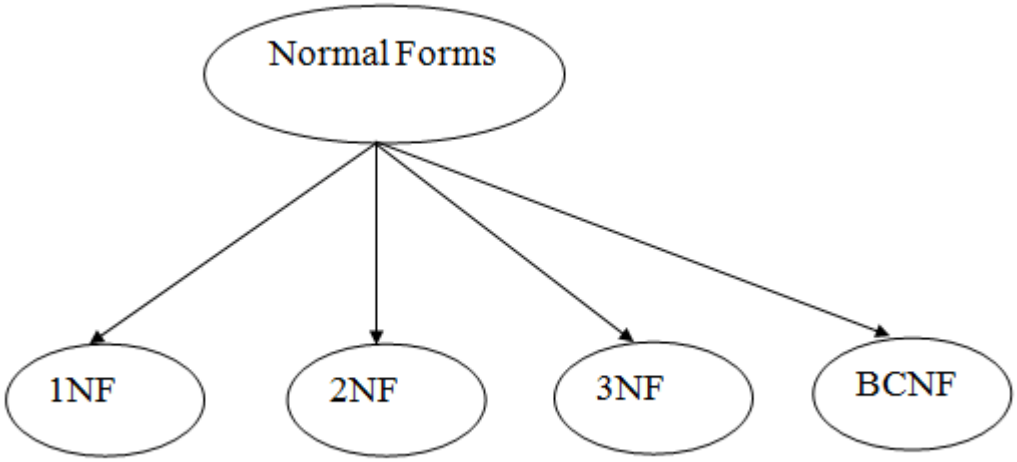
This is now in BCNF as in both the functional dependencies left side part is a key.

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:



Normal 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 multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains 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 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 the 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 atleast one of the following conditions for every non-trivial function 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:

1. {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 dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent 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

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 advance 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:

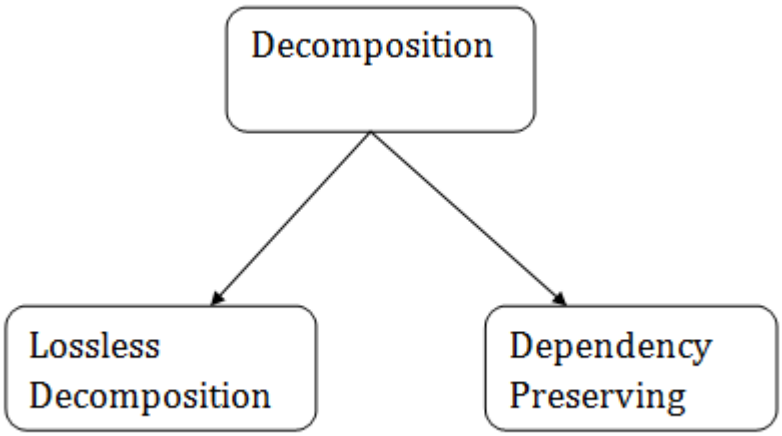
For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

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

Relational Decomposition

- When a relation in the relational model is not in appropriate normal form then the decomposition of a relation is required.
- In a database, it breaks the table into multiple tables.
- If the relation has no proper decomposition, then it may lead to problems like loss of information.
- Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.

Types of Decomposition



Lossless Decomposition

- If the information is not lost from the relation that is decomposed, then the decomposition will be lossless.
- The lossless decomposition guarantees that the join of relations will result in the same relation as it was decomposed.
- The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

Example:

EMPLOYEE_DEPARTMENT table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales
33	Alina	25	Delhi	438	Marketing

46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

The above relation is decomposed into two relations EMPLOYEE and DEPARTMENT

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY
22	Denim	28	Mumbai
33	Alina	25	Delhi
46	Stephan	30	Bangalore
52	Katherine	36	Mumbai
60	Jack	40	Noida

DEPARTMENT table

DEPT_ID	EMP_ID	DEPT_NAME
827	22	Sales
438	33	Marketing
869	46	Finance
575	52	Production
678	60	Testing

Now, when these two relations are joined on the common column "EMP_ID", then the resultant relation will look like:

Employee ⋈ Department

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales

33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

Hence, the decomposition is Lossless join decomposition.

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).

Multivalued Dependency

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.
- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.

Example: Suppose there is a bike manufacturer company which produces two colors(white and black) of each model every year.

BIKE_MODEL	MANUF_YEAR	COLOR
M2011	2008	White
M2001	2008	Black
M3001	2013	White
M3001	2013	Black
M4006	2017	White
M4006	2017	Black

Here columns COLOR and MANUF_YEAR are dependent on BIKE_MODEL and independent of each other.

In this case, these two columns can be called as multivalued dependent on BIKE_MODEL. The representation of these dependencies is shown below:

1. BIKE_MODEL → → MANUF_YEAR
2. BIKE_MODEL → → COLOR

This can be read as "BIKE_MODEL multidetermined MANUF_YEAR" and "BIKE_MODEL multidetermined COLOR".

join Dependency

- Join decomposition is a further generalization of Multivalued dependencies.
- If the join of R1 and R2 over C is equal to relation R, then we can say that a join dependency (JD) exists.
- Where R1 and R2 are the decompositions R1(A, B, C) and R2(C, D) of a given relations R (A, B, C, D).
- Alternatively, R1 and R2 are a lossless decomposition of R.
- A JD ⋈ {R1, R2,..., Rn} is said to hold over a relation R if R1, R2,....., Rn is a lossless-join decomposition.
- The *(A, B, C, D), (C, D) will be a JD of R if the join of join's attribute is equal to the relation R.
- Here, *(R1, R2, R3) is used to indicate that relation R1, R2, R3 and so on are a JD of R.

Inclusion Dependency

- Multivalued dependency and join dependency can be used to guide database design although they both are less common than functional dependencies.
- Inclusion dependencies are quite common. They typically show little influence on designing of the database.
- The inclusion dependency is a statement in which some columns of a relation are contained in other columns.
- The example of inclusion dependency is a foreign key. In one relation, the referring relation is contained in the primary key column(s) of the referenced relation.
- Suppose we have two relations R and S which was obtained by translating two entity sets such that every R entity is also an S entity.
- Inclusion dependency would be happen if projecting R on its key attributes yields a relation that is contained in the relation obtained by projecting S on its key attributes.
- In inclusion dependency, we should not split groups of attributes that participate in an inclusion dependency.
- In practice, most inclusion dependencies are key-based that is involved only keys.

Canonical Cover

In the case of updating the database, the responsibility of the system is to check whether the existing functional dependencies are getting violated during the process of updating. In case of a violation of functional dependencies in the new database state, the rollback of the system must take place.

A canonical cover or irreducible a set of functional dependencies FD is a simplified set of FD that has a similar closure as the original set FD.

Extraneous attributes

An attribute of an FD is said to be extraneous if we can remove it without changing the closure of the set of FD.

Example: Given a relational Schema R(A, B, C, D) and set of Function Dependency FD = { B → A, AD → BC, C → ABD }. Find the canonical cover?

Solution: Given FD = { B → A, AD → BC, C → ABD }, now decompose the FD using decomposition rule(Armstrong Axiom).

1. B → A
2. AD → B (using decomposition inference rule on AD → BC)
3. AD → C (using decomposition inference rule on AD → BC)
4. C → A (using decomposition inference rule on C → ABD)
5. C → B (using decomposition inference rule on C → ABD)
6. C → D (using decomposition inference rule on C → ABD)

Now set of FD = { B → A, AD → B, AD → C, C → A, C → B, C → D }

The next step is to find closure of the left side of each of the given FD by including that FD and excluding that FD, if closure in both cases are same then that FD is redundant and we remove that FD from the given set, otherwise if both the closures are different then we do not exclude that FD.

Calculating closure of all FD { B → A, AD → B, AD → C, C → A, C → B, C → D }

1a. Closure B+ = BA using FD = { **B → A**, AD → B, AD → C, C → A, C → B, C → D }

1b. Closure B+ = B using FD = { AD → B, AD → C, C → A, C → B, C → D }

From 1 a and 1 b, we found that both the Closure(by including **B → A** and excluding **B → A**) are not equivalent, hence FD B → A is important and cannot be removed from the set of FD.

2 a. Closure AD+ = ADBC using FD = { B →A, **AD → B**, AD → C, C → A, C → B, C → D }

2 b. Closure AD+ = ADCB using FD = { B → A, AD → C, C → A, C → B, C → D }

From 2 a and 2 b, we found that both the Closure (by including **AD → B** and excluding **AD → B**) are equivalent, hence FD **AD → B** is not important and can be removed from the set of FD.

Hence resultant FD = { B → A, AD → C, C → A, C → B, C → D }

3 a. Closure AD+ = ADCB using FD = { B →A, **AD → C**, C → A, C → B, C → D }

3 b. Closure AD+ = AD using FD = { B → A, C → A, C → B, C → D }

From 3 a and 3 b, we found that both the Closure (by including **AD → C** and excluding **AD → C**) are not equivalent, hence FD AD → C is important and cannot be removed from the set of FD.

Hence resultant FD = { B → A, AD → C, C → A, C → B, C → D }

4 a. Closure C+ = CABD using FD = { B →A, AD → C, **C → A**, C → B, C → D }

4 b. Closure C+ = CBDA using FD = { B → A, AD → C, C → B, C → D }

From 4 a and 4 b, we found that both the Closure (by including **C → A** and excluding **C → A**) are equivalent, hence FD **C → A** is not important and can be removed from the set of FD.

Hence resultant FD = { B → A, AD → C, C → B, C → D }

5 a. Closure C+ = CBDA using FD = { B →A, AD → C, **C → B**, C → D }

5 b. Closure C+ = CD using FD = { B → A, AD → C, C → D }

From 5 a and 5 b, we found that both the Closure (by including **C → B** and excluding **C → B**) are not equivalent, hence FD **C → B** is important and cannot be removed from the set of FD.

Hence resultant FD = { B → A, AD → C, C → B, C → D }

6 a. Closure C+ = CDBA using FD = { B →A, AD → C, C → B, **C → D** }

6 b. Closure C+ = CBA using FD = { B → A, AD → C, C → B }

From 6 a and 6 b, we found that both the Closure(by including **C → D** and excluding **C → D**) are not equivalent, hence FD **C → D** is important and cannot be removed from the set of FD.

Hence resultant FD = { B → A, AD → C, C → B, C → D }

- Since FD = { B → A, AD → C, C → B, C → D } is resultant FD, now we have checked the redundancy of attribute, since the left side of FD AD → C has two attributes, let's check their importance, i.e. whether they both are important or only one.

Closure AD+ = ADCB using FD = { B →A, **AD → C**, C → B, C → D }

Closure A+ = A using FD = { B →A, **AD → C**, C → B, C → D }

Closure D+ = D using FD = { B →A, **AD → C**, C → B, C → D }

Since the closure of AD+, A+, D+ that we found are not all equivalent, hence in FD AD → C, both A and D are important attributes and cannot be removed.

Hence resultant FD = { B → A, AD → C, C → B, C → D } and we can rewrite as

FD = { B → A, AD → C, C → BD } is Canonical Cover of FD = { B → A, AD → BC, C → ABD }.

Example 2: Given a relational Schema R(W, X, Y, Z) and set of Function Dependency FD = { W → X, Y → X, Z → WXY, WY → Z }. Find the canonical cover?

Solution: Given FD = { W → X, Y → X, Z → WXY, WY → Z }, now decompose the FD using decomposition rule(Armstrong Axiom).

1. W → X
2. Y → X
3. Z → W (using decomposition inference rule on Z → WXY)
4. Z → X (using decomposition inference rule on Z → WXY)
5. Z → Y (using decomposition inference rule on Z → WXY)
6. WY → Z

Now set of FD = { W → X, Y → X, WY → Z, Z → W, Z → X, Z → Y }

The next step is to find closure of the left side of each of the given FD by including that FD and excluding that FD, if closure in both cases are same then that FD is redundant and we remove that FD from the given set, otherwise if both the closures are different then we do not exclude that FD.

Calculating closure of all FD { W → X, Y → X, Z → W, Z → X, Z → Y, WY → Z }

1 a. Closure W+ = WX using FD = { **W → X**, Y → X, Z → W, Z → X, Z → Y, WY → Z }

1 b. Closure W+ = W using FD = { Y → X, Z → W, Z → X, Z → Y, WY → Z }

From 1 a and 1 b, we found that both the Closure (by including **W → X** and excluding **W → X**) are not equivalent, hence FD W → X is important and cannot be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → X, Z → Y, WY → Z }

2 a. Closure Y+ = YX using FD = { W → X, **Y → X**, Z → W, Z → X, Z → Y, WY → Z }

2 b. Closure Y+ = Y using FD = { W → X, Z → W, Z → X, Z → Y, WY → Z }

From 2 a and 2 b we found that both the Closure (by including **Y → X** and excluding **Y → X**) are not equivalent, hence FD Y → X is important and cannot be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → X, Z → Y, WY → Z }

3 a. Closure Z+ = ZWXY using FD = { W → X, Y → X, **Z → W**, Z → X, Z → Y, WY → Z }

3 b. Closure Z+ = ZXY using FD = { W → X, Y → X, Z → X, Z → Y, WY → Z }

From 3 a and 3 b, we found that both the Closure (by including **Z → W** and excluding **Z → W**) are not equivalent, hence FD Z → W is important and cannot be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → X, Z → Y, WY → Z }

4 a. Closure Z+ = ZXWY using FD = { W → X, Y → X, Z → W, **Z → X**, Z → Y, WY → Z }

4 b. Closure Z+ = ZWYX using FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

From 4 a and 4 b, we found that both the Closure (by including **Z → X** and excluding **Z → X**) are equivalent, hence FD Z → X is **not** important and can be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

5 a. Closure Z+ = ZYWX using FD = { W → X, Y → X, Z → W, **Z → Y**, WY → Z }

5 b. Closure Z+ = ZWX using FD = { W → X, Y → X, Z → W, WY → Z }

From 5 a and 5 b, we found that both the Closure (by including **Z → Y** and excluding **Z → Y**) are not equivalent, hence FD Z → X is important and cannot be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

6 a. Closure WY+ = WYZX using FD = { W → X, Y → X, Z → W, Z → Y, **WY → Z** }

6 b. Closure WY+ = WYX using FD = { W → X, Y → X, Z → W, Z → Y }

From 6 a and 6 b, we found that both the Closure (by including **WY → Z** and excluding **WY → Z**) are not equivalent, hence FD WY → Z is important and cannot be removed from the set of FD.

Hence resultant FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

Since FD = { W → X, Y → X, Z → W, Z → Y, WY → Z } is resultant FD now, we have checked the redundancy of attribute, since the left side of FD WY → Z has two attributes at its left, let's check their importance, i.e. whether they both are important or only one.

Closure WY+ = WYZX using FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

Closure W+ = WX using FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

Closure Y+ = YX using FD = { W → X, Y → X, Z → W, Z → Y, WY → Z }

Since the closure of WY+, W+, Y+ that we found are not all equivalent, hence in FD WY → Z, both W and Y are important attributes and cannot be removed.

Hence resultant FD = { W → X, Y → X, Z → W, Z → Y, WY → Z } and we can rewrite as:

FD = { W → X, Y → X, Z → WY, WY → Z } is Canonical Cover of FD = { W → X, Y → X, Z → WXY, WY → Z }.

Example 3: Given a relational Schema R(V, W, X, Y, Z) and set of Function Dependency FD = { V → W, VW → X, Y → VXZ }. Find the canonical cover?

Solution: Given FD = { V → W, VW → X, Y → VXZ }. now decompose the FD using decomposition rule (Armstrong Axiom).

1. V → W
2. VW → X
3. Y → V (using decomposition inference rule on Y → VXZ)
4. Y → X (using decomposition inference rule on Y → VXZ)
5. Y → Z (using decomposition inference rule on Y → VXZ)

Now set of FD = { V → W, VW → X, Y → V, Y → X, Y → Z }.

The next step is to find closure of the left side of each of the given FD by including that FD and excluding that FD, if closure in both cases are same then that FD is redundant and we remove that FD from the given set, otherwise if both the closures are different then we do not exclude that FD.

Calculating closure of all FD { V → W, VW → X, Y → V, Y → X, Y → Z }.

1 a. Closure V+ = VWX using FD = {**V → W**, VW → X, Y → V, Y → X, Y → Z}

1 b. Closure V+ = V using FD = {VW → X, Y → V, Y → X, Y → Z }

From 1 a and 1 b, we found that both the Closure(by including **V → W** and excluding **V → W**) are not equivalent, hence FD V → W is important and cannot be removed from the set of FD.

Hence resultant FD = { V → W, VW → X, Y → V, Y → X, Y → Z }.

2 a. Closure VW+ = VWX using FD = { V → W, **VW → X**, Y → V, Y → X, Y → Z }

2 b. Closure VW+ = VW using FD = { V → W, Y → V, Y → X, Y → Z }

From 2 a and 2 b, we found that both the Closure(by including **VW → X** and excluding **VW → X**) are not equivalent, hence FD VW → X is important and cannot be removed from the set of FD.

Hence resultant FD = { V → W, VW → X, Y → V, Y → X, Y → Z }.

3 a. Closure Y+ = YVXZW using FD = { V → W, VW → X, **Y → V**, Y → X, Y → Z }

3 b. Closure Y+ = YXZ using FD = { V → W, VW → X, Y → X, Y → Z }

From 3 a and 3 b, we found that both the Closure(by including **Y → V** and excluding **Y → V**) are not equivalent, hence FD Y → V is important and cannot be removed from the set of FD.

Hence resultant FD = { V → W, VW → X, Y → V, Y → X, Y → Z }.

4 a. Closure Y+ = YXVZW using FD = { V → W, VW → X, Y → V, **Y → X**, Y → Z }

4 b. Closure Y+ = YVZWX using FD = { V → W, VW → X, Y → V, Y → Z }

From 4 a and 4 b, we found that both the Closure(by including **Y → X** and excluding **Y → X**) are equivalent, hence FD Y → X is **not** important and can be removed from the set of FD.

Hence resultant FD = { V → W, VW → X, Y → V, Y → Z }.

5 a. Closure Y+ = YZVWX using FD = { V → W, VW → X, Y → V, **Y → Z** }

5 b. Closure Y+ = YVWX using FD = { V → W, VW → X, Y → V }

From 5 a and 5 b, we found that both the Closure(by including **Y → Z** and excluding **Y → Z**) are not equivalent, hence FD Y → Z is important and cannot be removed from the set of FD.

Hence resultant FD = { V → W, VW → X, Y → V, Y → Z }.

Since FD = { V → W, VW → X, Y → V, Y → Z } is resultant FD now, we have checked the redundancy of attribute, since the left side of FD VW → X has two attributes at its left, let's check their importance, i.e. whether they both are important or only one.

Closure VW+ = VWX using FD = { V → W, VW → X, Y → V, Y → Z }

Closure V+ = VWX using FD = { V → W, VW → X, Y → V, Y → Z }

Closure W+ = W using FD = { V → W, VW → X, Y → V, Y → Z }

Since the closure of VW+, V+, W+ we found that all the Closures of VW and V are equivalent, hence in FD VW → X, W is not at all an important attribute and can be removed.

Hence resultant FD = { V → W, V → X, Y → V, Y → Z } and we can rewrite as

FD = { V → WX, Y → VZ } is Canonical Cover of FD = { V → W, VW → X, Y → VXZ }.

CONCLUSION: From the above three examples we conclude that canonical cover / irreducible set of functional dependency follows the following steps, which we need to follow while calculating Canonical Cover.

STEP 1: For a given set of FD, decompose each FD using decomposition rule (Armstrong Axiom) if the right side of any FD has more than one attribute.

STEP 2: Now make a new set of FD having all decomposed FD.

STEP 3: Find closure of the left side of each of the given FD by including that FD and excluding that FD, if closure in both cases are same then that FD is redundant and we remove that FD from the given set, otherwise if both the closures are different then we do not exclude that FD.

STEP 4: Repeat step 4 till all the FDs in FD set are complete.

STEP 5: After STEP 4, find resultant FD = { B → A, AD → C, C → B, C → D } which are not redundant.

STEP 6: Check redundancy of attribute, by selecting those FD's from FD sets which are having more than one attribute on its left, let's an FD AD → C has two attributes at its left, let's check their importance, i.e. whether they both are important or only one.

STEP 6 a: Find Closure AD+

STEP 6 b: Find Closure A+

STEP 6 c: Find Closure D+

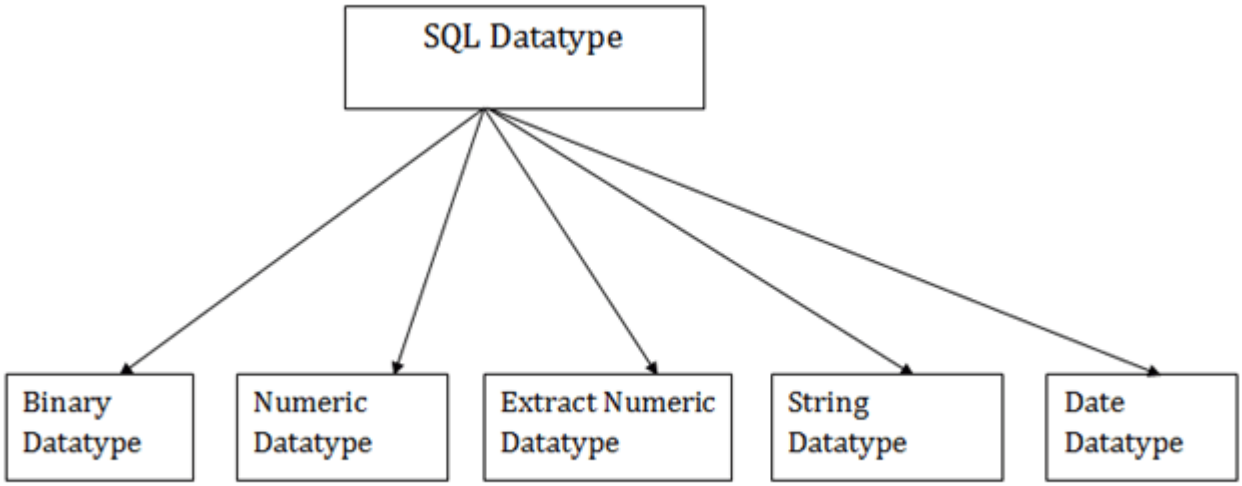
Compare Closure of STEP (6a, 6b, 6c) if the closure of AD+, A+, D+ are not equivalent, hence in FD AD → C, both A and D are important attributes and cannot be removed, otherwise, we remove the redundant attribute.

Unit 4

SQL Datatype

- SQL Datatype is used to define the values that a column can contain.
- Every column is required to have a name and data type in the database table.

Datatype of SQL:



1. Binary Datatypes

There are Three types of binary Datatypes which are given below:

Data Type	Description
binary	It has a maximum length of 8000 bytes. It contains fixed-length binary data.
varbinary	It has a maximum length of 8000 bytes. It contains variable-length binary data.
image	It has a maximum length of 2,147,483,647 bytes. It contains variable-length binary data.

2. Approximate Numeric Datatype :

The subtypes are given below:

Data type	From	To	Description
float	-1.79E + 308	1.79E + 308	It is used to specify a floating-point value e.g. 6.2, 2.9 etc.

real	-3.40e + 38	3.40E + 38	It specifies a single precision floating point number

3. Exact Numeric Datatype

The subtypes are given below:

Data type	Description
int	It is used to specify an integer value.
smallint	It is used to specify small integer value.
bit	It has the number of bits to store.
decimal	It specifies a numeric value that can have a decimal number.
numeric	It is used to specify a numeric value.

4. Character String Datatype

The subtypes are given below:

Data type	Description
char	It has a maximum length of 8000 characters. It contains Fixed-length non-unicode characters.
varchar	It has a maximum length of 8000 characters. It contains variable-length non-unicode characters.
text	It has a maximum length of 2,147,483,647 characters. It contains variable-length non-unicode characters.

5. Date and time Datatypes

The subtypes are given below:

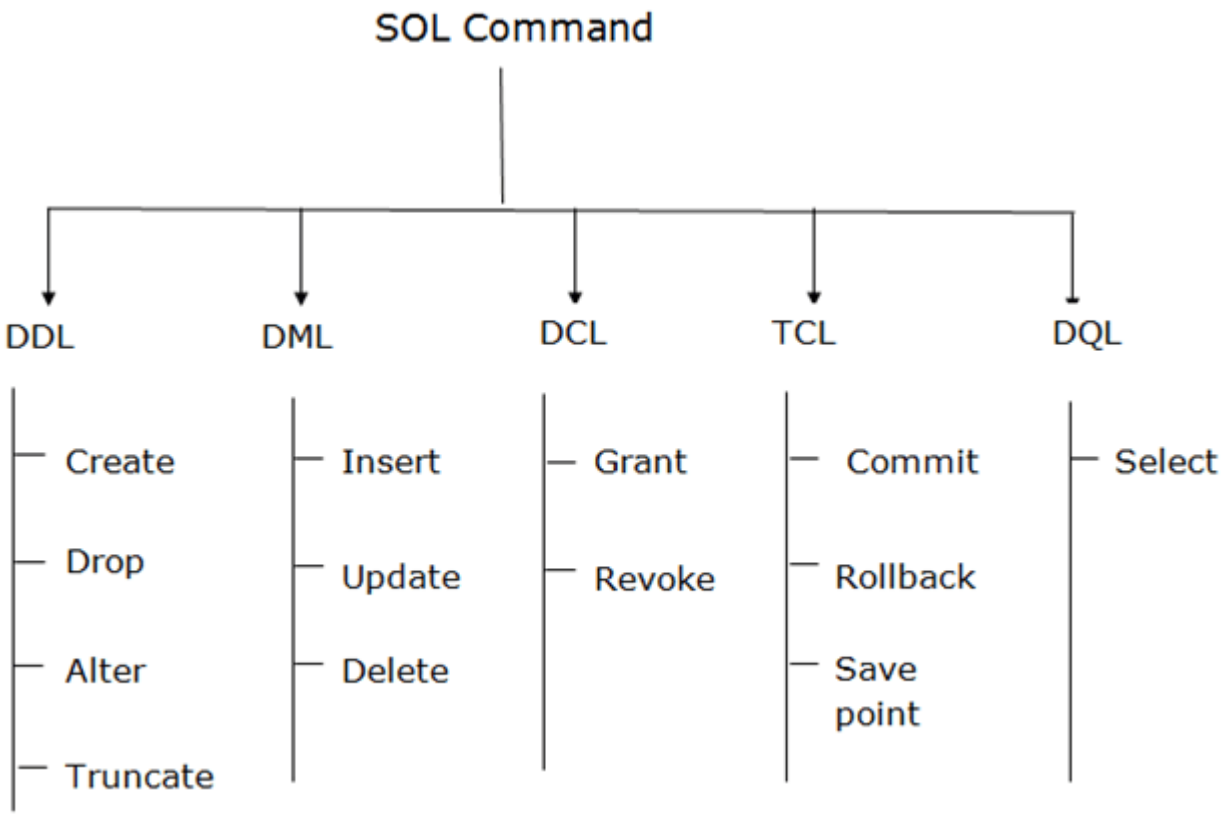
Datatype	Description
date	It is used to store the year, month, and days value.
time	It is used to store the hour, minute, and second values.
timestamp	It stores the year, month, day, hour, minute, and the second value.

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

a. 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);

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

Syntax

1. DROP TABLE ;

Example

1. DROP TABLE EMPLOYEE;

c. 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));

d. 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 rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. 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");
```

b. 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] [W
   HERE CONDITION]
```

For example:

```
1. UPDATE students
2. SET User_Name = 'Sonoo'
3. WHERE Student_Id = '3'
```

c. 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

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

Example

```
1. GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USE
   R;
```

b. 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

a. 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;
```

b. 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;
```

c. 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:

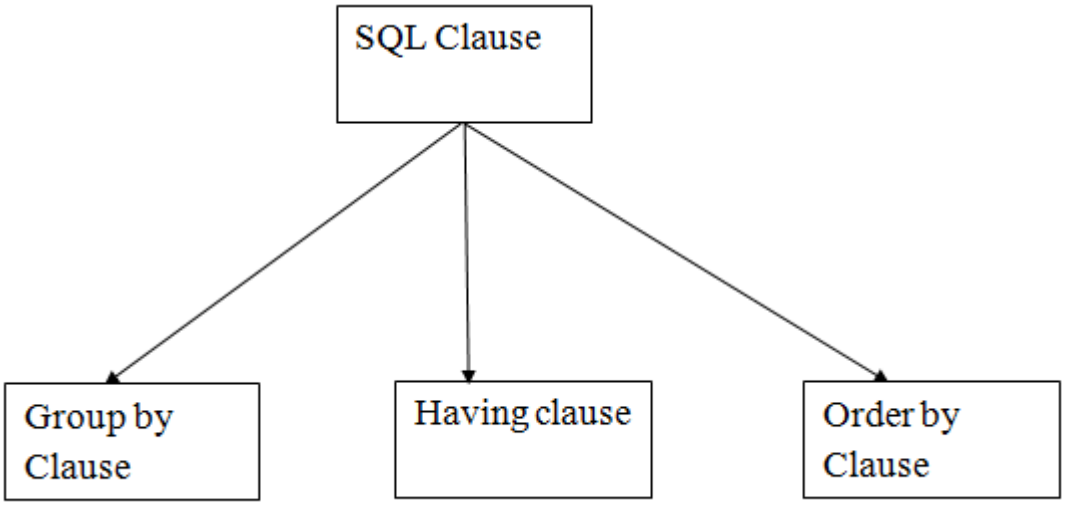
```
1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;
```

For example:

```
1. SELECT emp_name
2. FROM employee
3. WHERE age > 20;
```

4.3 SQL Clauses

The following are the various SQL clauses:



1. GROUP BY

- SQL GROUP BY statement is used to arrange identical data into groups. The GROUP BY statement is used with the SQL SELECT statement.
- The GROUP BY statement follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.
- The GROUP BY statement is used with aggregation function.

Syntax

1. SELECT column
2. FROM table_name
3. WHERE conditions
4. GROUP BY column
5. ORDER BY column

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30

Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example:

```
1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY;
```

Output:

```
Com1    5
Com2    3
Com3    2
```

2. HAVING

- HAVING clause is used to specify a search condition for a group or an aggregate.
- Having is used in a GROUP BY clause. If you are not using GROUP BY clause then you can use HAVING function like a WHERE clause.

Syntax:

```
1. SELECT column1, column2
2. FROM table_name
3. WHERE conditions
4. GROUP BY column1, column2
5. HAVING conditions
6. ORDER BY column1, column2;
```

Example:

```
1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(*)>2;
```

Output:

```
Com1    5
Com2    3
```

3. ORDER BY

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

Syntax:

```
1. SELECT column1, column2
2. FROM table_name
3. WHERE condition
4. ORDER BY column1, column2... ASC|DESC;
```

Where

ASC: It is used to sort the result set in ascending order by expression.

DESC: It sorts the result set in descending order by expression.

Example: Sorting Results in Ascending Order

Table:

CUSTOMER

CUSTOMER_ID	NAME	ADDRESS
12	Kathrin	US
23	David	Bangkok
34	Alina	Dubai
45	John	UK
56	Harry	US

Enter the following SQL statement:

```
1. SELECT *
2. FROM CUSTOMER
3. ORDER BY NAME;
```

Output:

CUSTOMER_ID	NAME	ADDRESS
34	Alina	Dubai
23	David	Bangkok
56	Harry	US
45	John	UK
12	Kathrin	US

Example: Sorting Results in Descending Order

Using the above CUSTOMER table

```
1. SELECT *
2. FROM CUSTOMER
3. ORDER BY NAME DESC;
```

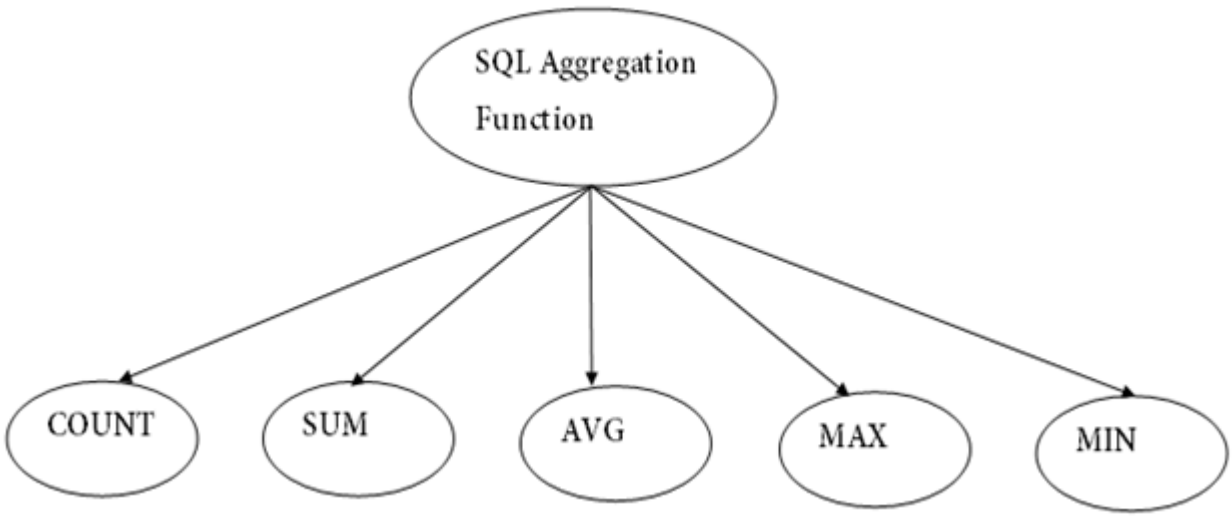
Output:

CUSTOMER_ID	NAME	ADDRESS
12	Kathrin	US
45	John	UK
56	Harry	US
23	David	Bangkok
34	Alina	Dubai

SQL Aggregate Functions

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Types of SQL Aggregation Function



1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

1. COUNT(*)
2. or
3. COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Com1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

```
1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;
```

Output:

```
10
```

Example: COUNT with WHERE

```
1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;
3. WHERE RATE>=20;
```

Output:

```
7
```

Example: COUNT() with DISTINCT

```
1. SELECT COUNT(DISTINCT COMPANY)
2. FROM PRODUCT_MAST;
```

Output:

```
3
```

Example: COUNT() with GROUP BY

```
1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
```

3. GROUP BY COMPANY;

Output:

```
Com1    5
Com2    3
Com3    2
```

Example: COUNT() with HAVING

1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(*)>2;

Output:

```
Com1    5
Com2    3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

1. SUM()
2. or
3. SUM([ALL|DISTINCT] expression)

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

4. **Example: SUM()**

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST;

Output:

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	**Com3**	**5**	**10**	**50**
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	**Com1**	**5**	**30**	**150**
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	**Com3**	**4**	**30**	**120**

```
670
```

Example: SUM() with WHERE

```
1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3;
```

Output:

```
320
```

Example: SUM() with GROUP BY

```
1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3
4. GROUP BY COMPANY;
```

Output:

```
Com1    150
Com3    170
```

Example: SUM() with HAVING

```
1. SELECT COMPANY, SUM(COST)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING SUM(COST)>=70;
```

Output:

```
Com1    125
Com2    75
Com3    120
```

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

1. AVG()
2. or
3. AVG([ALL|DISTINCT] expression)

Example:

```
1. SELECT AVG(COST)
2. FROM PRODUCT_MAST;
```

Output:

```
67.00
```

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

1. MAX()
2. or
3. MAX([ALL|DISTINCT] expression)

Example:

```
1. SELECT MAX(RATE)
2. FROM PRODUCT_MAST;
```

```
30
```

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

1. MIN()

2. or
3. MIN([ALL|DISTINCT] expression)

Example:

```
1. SELECT MIN(RATE)
2. FROM PRODUCT_MAST;
```

Output:

```
10
```

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
3. RIGHT JOIN
4. 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 angels	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

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

1. SELECT column_name FROM table1
2. UNION
3. SELECT column_name FROM table2;

Example:

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
1	**jack**
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
1. SELECT * FROM First
2. UNION
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
1. SELECT column_name FROM table1
2. UNION ALL
3. SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
1. SELECT * FROM First
2. UNION ALL
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

```
1. SELECT column_name FROM table1
2. INTERSECT
3. SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

Intersect query will be:

```
1. SELECT * FROM First
2. INTERSECT
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
1. SELECT column_name FROM table1
2. MINUS
3. SELECT column_name FROM table2;
```

Example

Using the above First and Second table.

Minus query will be:

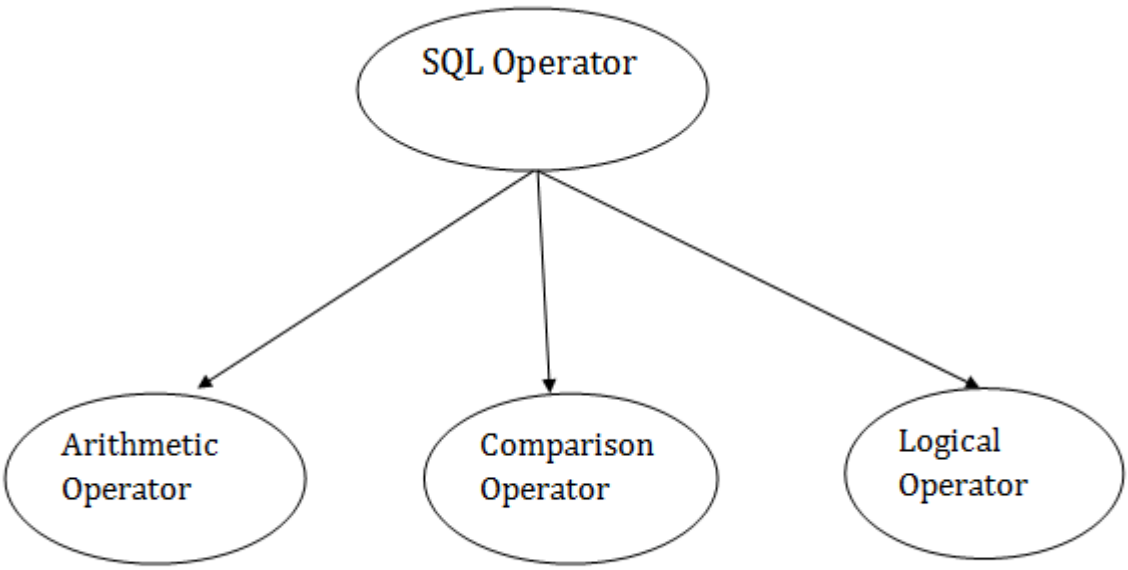
```
1. SELECT * FROM First
2. MINUS
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

SQL Operator

There are various types of SQL operator:



SQL Arithmetic Operators

Let's assume 'variable a' and 'variable b'. Here, 'a' contains 20 and 'b' contains 10.

Operator	Description	Example
+	It adds the value of both operands.	a+b will give 30
-	It is used to subtract the right-hand operand from the left-hand operand.	a-b will give 10
*	It is used to multiply the value of both operands.	a*b will give 200
/	It is used to divide the left-hand operand by the right-hand operand.	a/b will give 2
%	It is used to divide the left-hand operand by the right-hand operand and returns reminder.	a%b will give 0

SQL Comparison Operators:

Let's assume 'variable a' and 'variable b'. Here, 'a' contains 20 and 'b' contains 10.

Operator	Description	Example
=	It checks if two operands values are equal or not, if the values are queal then condition becomes true.	(a=b) is not true
!=	It checks if two operands values are equal or not, if values are not equal, then condition becomes true.	(a!=b) is true

<>	It checks if two operands values are equal or not, if values are not equal then condition becomes true.	(a<>b) is true
>	It checks if the left operand value is greater than right operand value, if yes then condition becomes true.	(a>b) is not true
<	It checks if the left operand value is less than right operand value, if yes then condition becomes true.	(a<b) is true
>=	It checks if the left operand value is greater than or equal to the right operand value, if yes then condition becomes true.	(a>=b) is not true
<=	It checks if the left operand value is less than or equal to the right operand value, if yes then condition becomes true.	(a<=b) is true
!<	It checks if the left operand value is not less than the right operand value, if yes then condition becomes true.	(a!=b) is not true
!>	It checks if the left operand value is not greater than the right operand value, if yes then condition becomes true.	(a!>b) is true

SQL Logical Operators

There is the list of logical operator used in SQL:

Operator	Description
ALL	It compares a value to all values in another value set.
AND	It allows the existence of multiple conditions in an SQL statement.
ANY	It compares the values in the list according to the condition.
BETWEEN	It is used to search for values that are within a set of values.
IN	It compares a value to that specified list value.
NOT	It reverses the meaning of any logical operator.
OR	It combines multiple conditions in SQL statements.
EXISTS	It is used to search for the presence of a row in a specified table.
LIKE	It compares a value to similar values using wildcard operator.

Views in SQL

- Views in SQL are considered as a virtual table. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.

Sample table:

Student_Detail

STU_ID	NAME	ADDRESS

1	Stephan	Delhi
2	Kathrin	Noida
3	David	Ghaziabad
4	Alina	Gurugram

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	18
4	Alina	90	20
5	John	96	18

1. Creating view

A view can be created using the **CREATE VIEW** statement. We can create a view from a single table or multiple tables.

Syntax:

```
1. CREATE VIEW view_name AS
2. SELECT column1, column2.....
3. FROM table_name
4. WHERE condition;
```

2. Creating View from a single table

In this example, we create a View named DetailsView from the table Student_Detail.

Query:

```
1. CREATE VIEW DetailsView AS
2. SELECT NAME, ADDRESS
3. FROM Student_Details
4. WHERE STU_ID < 4;
```

Just like table query, we can query the view to view the data.

```
1. SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida

David	Ghaziabad

3. Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

Query:

```
1. CREATE VIEW MarksView AS
2. SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARK
   S
3. FROM Student_Detail, Student_Mark
4. WHERE Student_Detail.NAME = Student_Marks.NAME;
```

To display data of View MarksView:

```
1. SELECT * FROM MarksView;
```

NAME	ADDRESS	MARKS
Stephan	Delhi	97
Kathrin	Noida	86
David	Ghaziabad	74
Alina	Gurugram	90

4. Deleting View

A view can be deleted using the Drop View statement.

Syntax

```
1. DROP VIEW view_name;
```

Example:

If we want to delete the View **MarksView**, we can do this as:

```
1. DROP VIEW MarksView;
```

SQL Index

- Indexes are special lookup tables. It is used to retrieve data from the database very fast.

- An Index is used to speed up select queries and where clauses. But it shows down the data input with insert and update statements. Indexes can be created or dropped without affecting the data.
- An index in a database is just like an index in the back of a book.
- **For example:** When you reference all pages in a book that discusses a certain topic, you first have to refer to the index, which alphabetically lists all the topics and then referred to one or more specific page numbers.

1. Create Index statement

It is used to create an index on a table. It allows duplicate value.

Syntax

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Example

```
CREATE INDEX idx_name
ON Persons (LastName, FirstName);
```

2. Unique Index statement

It is used to create a unique index on a table. It does not allow duplicate value.

Syntax

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

Example

```
CREATE UNIQUE INDEX websites_idx
ON websites (site_name);
```

3. Drop Index Statement

It is used to delete an index in a table.

Syntax

```
DROP INDEX index_name;
```

Example

```
DROP INDEX websites_idx;
```

SQL Sub Query

A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

```
1. SELECT column_name
2. FROM table_name
3. WHERE column_name expression operator
4. ( SELECT column_name  from table_name WHERE ... );
```

Example

Consider the EMPLOYEE table have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be:

```
1. SELECT *
2.     FROM EMPLOYEE
3.     WHERE ID IN (SELECT ID
4.     FROM EMPLOYEE
```

```
5.  WHERE SALARY > 4500);
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

2. Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

```
1. INSERT INTO table_name (column1, column2, column3....)
2. SELECT *
3. FROM table_name
4. WHERE VALUE OPERATOR
```

Example

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
1. INSERT INTO EMPLOYEE_BKP
2.    SELECT * FROM EMPLOYEE
3.    WHERE ID IN (SELECT ID
4.    FROM EMPLOYEE);
```

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

```
1. UPDATE table
2. SET column_name = new_value
3. WHERE VALUE OPERATOR
4.    (SELECT COLUMN_NAME
5.    FROM TABLE_NAME
```

```
6.  WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
1.  UPDATE EMPLOYEE
2.     SET SALARY = SALARY * 0.25
3.     WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
4.        WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

```
1.  DELETE FROM TABLE_NAME
2.  WHERE VALUE OPERATOR
3.     (SELECT COLUMN_NAME
4.     FROM TABLE_NAME
5.     WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

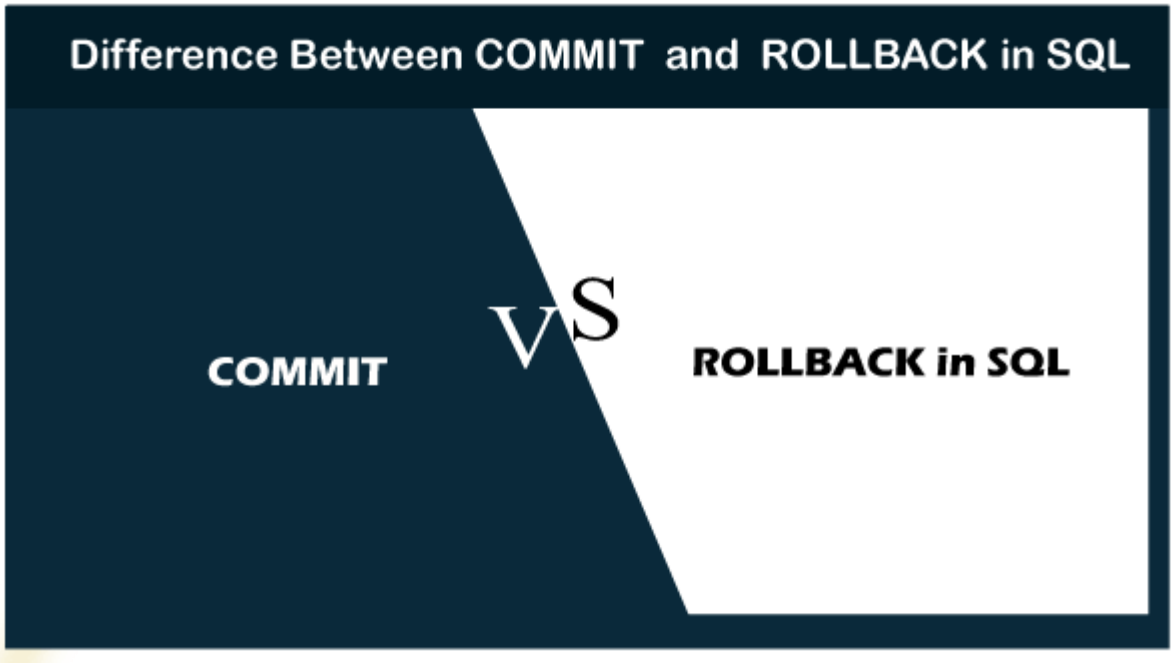
```
1.  DELETE FROM EMPLOYEE
2.     WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
3.        WHERE AGE >= 29 );
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

Difference between COMMIT and ROLLBACK in SQL

COMMIT and ROLLBACK are the two terms used in the transactional statement to perform or undo the SQL transaction. Before going to the term COMMIT and ROLLBACK, we need to understand the term **Transaction**. A transaction is a logical term with some sequence of instructions or queries to make a complete transaction execution. Each transaction starts with a specific task and ends when the group of tasks is completed. If any of the tasks fails, the transaction is a failure. To make a complete transaction in SQL, we need to perform various activities such as: starts the transaction, set the transaction, commit, the ROLLBACK, and the SAVEPOINT of the transaction. Here we will discuss only two terms, COMMIT and ROLLBACK, and their differences in SQL.



What is SQL COMMIT?

A COMMIT is the SQL command used **in the transaction tables or database** to make the current transaction or database **statement** as permanent. It shows the successful completion of a transaction. If we have successfully executed the transaction statement or a simple database query, we want to make the changes permanent. We need to perform the **commit command** to save the changes, and these changes become permanent for all users. Furthermore, once the commit command is executed in the database, we cannot regain its previous states in which it was earlier before the execution of the first statement.

Syntax

```
1. COMMIT;
```

Consider the Employees tables that containing the following records:

EMPID	EMP NAME	AGE	ADDRESS	SALARY
101	JOHN	32	CALIFORNIA	5000$
102	DANIEL	38	LOS ANGELES	2900$
103	MARIA	27	TEXAS	4900$
104	JOY	39	FLORIDA	3200$
105	ROOT	28	SAN FRANCISCO	2800$
106	MARCUS	27	CALIFORNIA	1000$

In the above example, we will delete all those records from the **EMPLOYEES** table whose age is 27 and then **COMMIT** query to make the changes as permanent that visible for all users in the database records.

Follow the given below SQL query:

```
1. DELETE from Employees Where age = 27;
2. COMMIT;
```

After that, use the **select** command to fetch all the records from the Employees table.

```
1. Select * from Employees;
```

Output

EMPID	EMP NAME	AGE	ADDRESS	SALARY

101	JOHN	32	CALIFORNIA	5000$
102	DANIEL	38	LOS ANGELES	2900$
104	JOY	39	FLORIDA	3200$
105	ROOT	28	SAN FRANCISCO	2800$

In the above table, when the COMMIT query is executed, it permanently saves the EMPLOYEES table changes, and these changes visible to all database users.

What is SQL ROLLBACK?

The **SQL ROLLBACK** command is used to roll back the current transaction state if any error occurred during the execution of a transaction. In a transaction, the error can be a system failure, power outage, incorrect execution of the transaction, system crash, etc. Generally, a rollback command performs the current transaction's rollback action to return the transaction on its **previous state** or the first statement. A rollback command can only be executed if the user has not performed the **COMMIT** command on the current transaction or statement.

Syntax

```
1. ROLLBACK;
```

Consider the Employees tables that containing the following records:

EMPID	EMP NAME	AGE	ADDRESS	SALARY
101	JOHN	32	CALIFORNIA	5000$
102	DANIEL	38	LOS ANGELES	2900$
103	MARIA	27	TEXAS	4900$
104	JOY	39	FLORIDA	3200$
105	ROOT	28	SAN FRANCISCO	2800$
106	MARCUS	27	CALIFORNIA	1000$

In the above example, we will delete all those records from the **EMPLOYEES** table whose age is 27 and then perform the **ROLLBACK** query to retrieve the deleted records from the EMPLOYEES table.

Follow the given below SQL query:

1. DELETE from Employees Where age = 27;
2. ROLLBACK;

TABLE - EMPLOYEES

EMPID	EMP NAME	AGE	ADDRESS	SALARY
101	JOHN	32	CALIFORNIA	5000$
102	DANIEL	38	LOS ANGELES	2900$
104	JOY	39	FLORIDA	3200$
105	ROOT	28	SAN FRANCISCO	2800$

Here, in the above table, there are two rows whose age is 27 deleted from the Employees table that satisfy the age condition. And then ROLLBACK query to undo the operations.

After that, use the **select** command to retrieve all the records from the Employees table.

1. Select * from Employees;

Output

TABLE - EMPLOYEES

EMPID	EMP NAME	AGE	ADDRESS	SALARY
101	JOHN	32	CALIFORNIA	5000$
102	DANIEL	38	LOS ANGELES	2900$
103	MARIA	27	TEXAS	4900$
104	JOY	39	FLORIDA	3200$
105	ROOT	28	SAN FRANCISCO	2800$
106	MARCUS	27	CALIFORNIA	1000$

Difference between the COMMIT and ROLLBACK

Comparison Based on Parameters	COMMIT Statement	ROLLBACK Statement

Definition/ Basic	A COMMIT statement is used to save the changes on the current transaction is permanent.	A Rollback statement is used to undo all the changes made on the current transaction.
Transaction condition	Once the current transaction is completely executed using the COMMIT command, it can't undo its previous state.	Whereas in the Rollback statement, once the current transaction is successfully executed, it can reach its previous state using the ROLLBACK command.
Syntax of Statement	Commit;	Rollback;
Occurrence	The COMMIT statement is applied when the transaction is completed.	The Rollback statement occurs when the transaction is either aborted, power failure, or incorrect execution of system failure.
Successfully executed the statement.	If all the statements are executed successfully without any error, the COMMIT statement will permanently save the state.	If any operations fail during the completion of a transaction, it shows all the changes have not been successfully executed, and we can undo them using the ROLLBACK statement.
Visible change	When we perform the commit command, the current transaction statement becomes permanent and visible to all users.	Whereas the rollback command is also visible to all users, even the current transaction may contain the wrong or right information.

Conclusion

The commit command ensures the transaction changes are permanently saved in the database or tables to complete a transaction. In contrast, the rollback command is used to undo all changes during the transaction for occurring any types of issues such as power failure, wrong data, or return the current transaction to its initial phase.

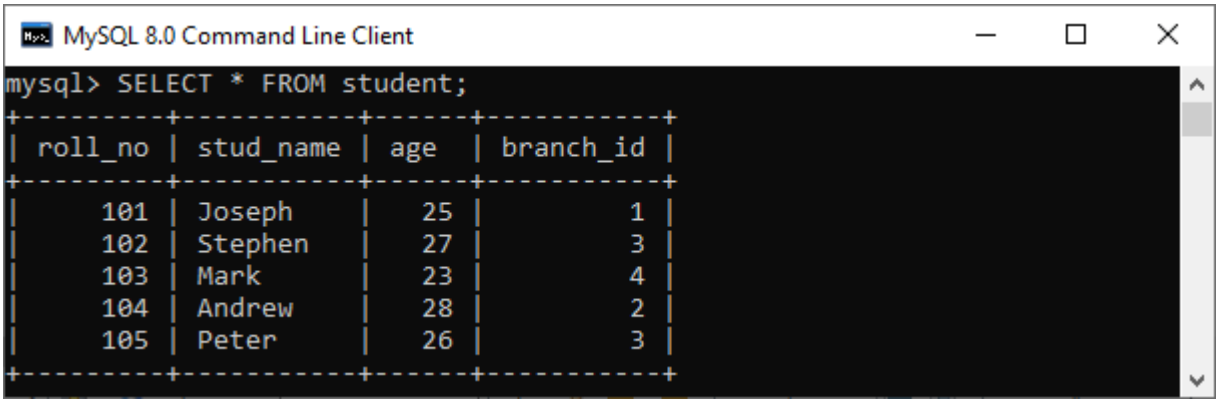
Denormalization in Databases

When we normalize tables, we break them into multiple smaller tables. So when we want to retrieve data from multiple tables, we need to perform some kind of join operation on them. In that case, we use the denormalization technique that eliminates the drawback of normalization.

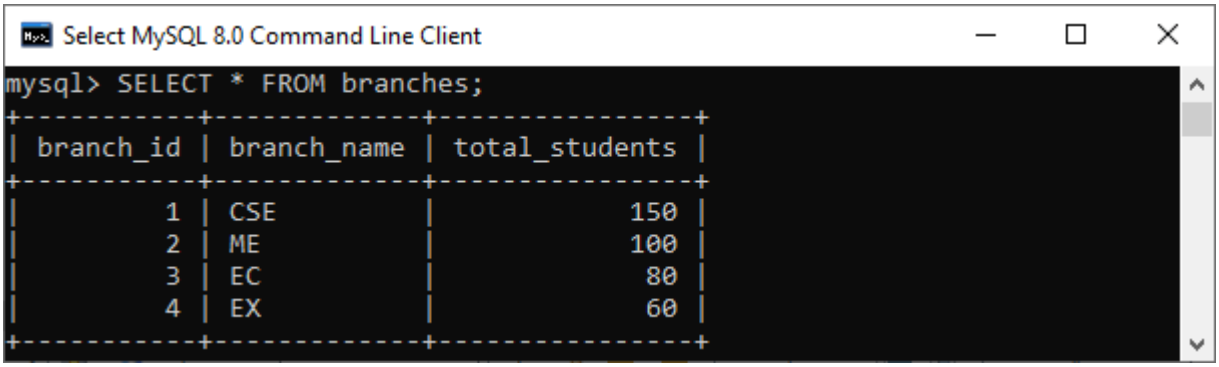
Denormalization is a technique used by database administrators to optimize the efficiency of their database infrastructure. This method allows us to add redundant data into a normalized database to alleviate issues with database queries that merge data from several tables into a single table. The denormalization concept is based on the definition of normalization that is defined as arranging a database into tables correctly for a particular purpose.

NOTE: Denormalization does not indicate not doing normalization. It is an optimization strategy that is used after normalization has been achieved.

For Example, We have two table students and branch after performing normalization. The student table has the attributes roll_no, stud-name, age, and branch_id.



Additionally, the branch table is related to the student table with branch_id as the student table's foreign key.



A JOIN operation between these two tables is needed when we need to retrieve all student names as well as the branch name. Suppose we want to change the student name only, then it is great if the table is small. The issue here is that if the tables are big, joins on tables can take an excessively long time.

In this case, we'll update the database with denormalization, redundancy, and extra effort to maximize the efficiency benefits of fewer joins. Therefore, we can add the branch name's data from the Branch table to the student table and optimizing the database.

Pros of Denormalization

The following are the advantages of denormalization:

1. Enhance Query Performance

Fetching queries in a normalized database generally requires joining a large number of tables, but we already know that the more joins, the slower the query. To overcome this, we can add

redundancy to a database by copying values between parent and child tables, minimizing the number of joins needed for a query.

2. Make database more convenient to manage

A normalized database is not required calculated values for applications. Calculating these values on-the-fly will take a longer time, slowing down the execution of the query. Thus, in denormalization, fetching queries can be simpler because we need to look at fewer tables.

3. Facilitate and accelerate reporting

Suppose you need certain statistics very frequently. It requires a long time to create them from live data and slows down the entire system. Suppose you want to monitor client revenues over a certain year for any or all clients. Generating such reports from live data will require "searching" throughout the entire database, significantly slowing it down.

Cons of Denormalization

The following are the disadvantages of denormalization:

- It takes large storage due to data redundancy.
- It makes it expensive to updates and inserts data in a table.
- It makes update and inserts code harder to write.
- Since data can be modified in several ways, it makes data inconsistent. Hence, we'll need to update every piece of duplicate data. It's also used to measure values and produce reports. We can do this by using triggers, transactions, and/or procedures for all operations that must be performed together.

How is denormalization different from normalization?

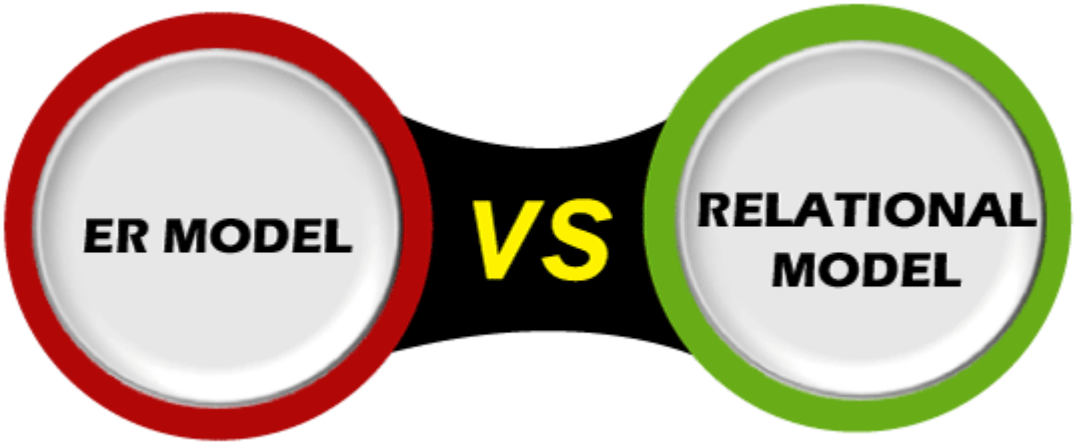
The denormalization is different from normalization in the following manner:

- Denormalization is a technique used to merge data from multiple tables into a single table that can be queried quickly. Normalization, on the other hand, is used to delete redundant data from a database and replace it with non-redundant and reliable data.
- Denormalization is used when joins are costly, and queries are run regularly on the tables. Normalization, on the other hand, is typically used when a large number of insert/update/delete operations are performed, and joins between those tables are not expensive.

Difference between ER Model and Relational Model

The E-R Model and Relational Model are two aspects of the Data Model in DBMS that are used to construct databases at the physical, logical, and view levels. This article explains the complete overview of the E-R Model and Relational Model. The difference between these models is the most common part of an interview question. **The key distinction is that the E-**

R Model is entity-specific, while the Relational Model is table-specific. Before making the comparison, we will first know these Data Models.



What is ER Model?

An ER model stands for the Entity-Relationship model that **Peter Chen developed in 1976**. This model consists of a collection of entities (Real word objects) and their relationships. It describes the database's conceptual view. We must ensure that no two entities are identical in this context.

ER Model describes the system's logical view from a data perspective formed by the entity set, relationship set, and attributes. In this model, all entities come under the entity set, all relations between the entities come under the relationship set, and attributes describe the properties of entities.

Let us understand ER model with an example. Suppose we have **two real-world entities** named **Student** and **Branch** that will further form an Entity set. Now we can easily form a relation between them as the **Student belongs to a Branch**. It shows how we can get a relationship set from ER Model. Noted that the ER Model content must conform to constraints like mapping cardinality. Finally, the attributes of these entities would be:

For Student: stud_id, stud_name, address, mobile, mail-id.

For Branch: branch_id, branch_name, num_of_stud.

What is the Relational Model?

In 1970, **E.F. Codd** developed the relational model. He proposed this model as well as a non-procedural approach for modeling data in the form of relations or tables. In the Relational Model, tables are usually interpreted as relations. If we model the database using ER diagrams, we must convert them into the relational model, which can be implemented by one of the RDBMS languages such as SQL and MySQL.

In the relational model, each table contains rows and columns where we can have any number of rows, but the number of columns must be definite. The table rows are called tuples that include the complete information about specific entities. Records are a set of tuples, so the Relational model is also known as the Record-based Model. Table columns are called attributes because they characterize a table's properties. To store values, each attribute must have a type.

Key Differences between ER Model and Relational Model

The following points explain the main differences between ER Model and Relational Model:

- The main distinction between the ER model and the Relational Model is that the ER model describes the relationship between entities and their attributes. On the other hand, the Relational Model referred to the implementation of our model.
- The Relational Model is the implementation or representational model, while the ER Model is the high-level or conceptual model.
- The data in components such as entity sets, relationship sets, and attributes are represented by an ER model. The Relational model, on the other hand, defines data in components such as tuples, attributes, and attribute domains.
- As compared to a Relational Model, an ER model makes it easier to understand the relationships between entities.
- Mapping Cardinality is always a constraint in the ER model, while the cardinality constraint cannot be defined in the Relational Model.

ER Model vs. Relational Model Comparison Chart

The following comparison chart explains their main differences in a quick manner:

Comparison Basis	ER Model	Relational Model
Basic	It's used to describe a set of objects known as entities, as well as the relationships between them.	It's used to represent a collection of tables as well as the relationships between them.
Type	It is a high-level or conceptual model.	It is the implementation or representational model.
Components	It represents components as Entity, Entity Type, and Entity Set.	It represents components as domain, attributes, and tuples.
Used By	This model is helpful for those people who don't have any knowledge about how data is implemented.	This model is mostly famous among programmers.

Relationship	It is easy to understand the relationship between entities.	Compared to ER model, it is easier to derive the relation between tables in the Relational model.
Mapping	This model explains how to map Cardinalities. The uniqueness of data values in a row is referred to as cardinality.	This model does not describe mapping cardinalities.

Conclusion

In this article, we have made a comparison between ER Model and Relational Model. We may conclude that if we convert an E-R model to a Relational model, we must ensure that each strong entity has its own table.

Unit 5

Introduction of PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are facts about PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.

Features of PL/SQL

PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL\SQL execution environment https://www.tutorialspoint.com/plsql/plsql_environment_setup.htm

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL –

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
******	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators

<>, '=, ~=, ^= Different versions of NOT EQUAL

PL/SQL Block Structure

In PL/SQL, as in most other procedural languages, the smallest meaningful grouping of code is known as a *block*. A block is a unit of code that provides execution and scoping boundaries for variable declarations and exception handling. PL/SQL allows you to create *anonymous blocks* (blocks of code that have no name) and *named blocks*, which may be packages, procedures, functions, triggers, or object types.

A PL/SQL block has four different sections:

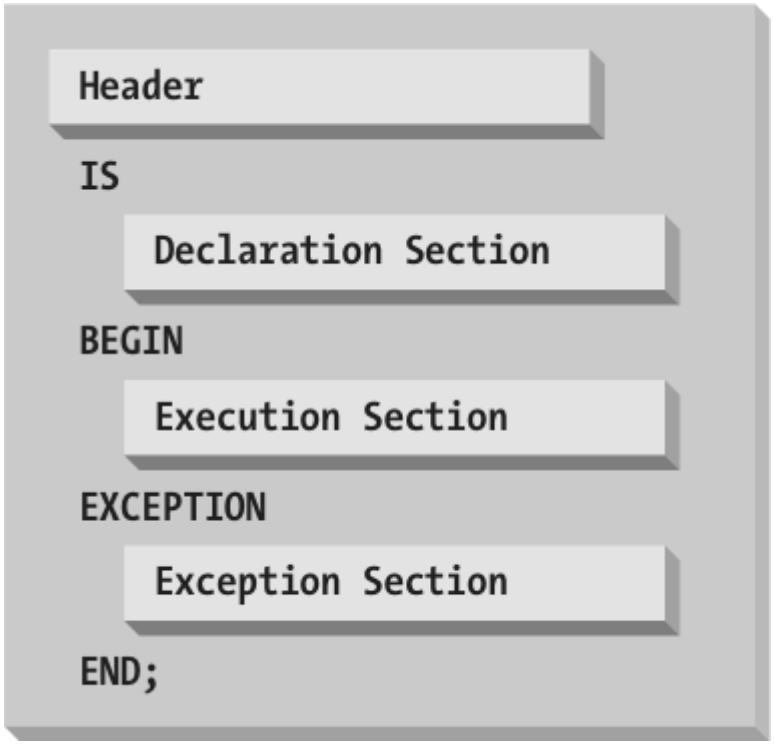


Figure 3-1. The PL/SQL block structure

Header

Used only for named blocks. The header determines the way the named block or program must be called. Optional.

Declaration section

Identifies variables, cursors, and subblocks that are referenced in the execution and exception sections. Optional.

Execution section

Statements the PL/SQL runtime engine will execute at runtime. Mandatory.

Exception section

Handles exceptions to normal processing (warnings and error conditions). This is an optional section of the PL/SQL blocks.

The Keyword 'END' marks the end of PL/SQL block.

PL/SQL - Data Types

FLOAT - floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

INT/ INTEGER - integer type with maximum precision of 38 decimal digits

REAL - Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

CHAR - Fixed-length character string with maximum size of 32,767 bytes

VARCHAR2 - Variable-length character string with maximum size of 32,767 bytes

LONG - Variable-length character string with maximum size of 32,760 bytes

Date and Time: This data type is used to store fixed data type which displays and saves time and date values. The default data format saved into the database might be 'DD-MM-YY'.

Boolean: These include logical values on which logical operations are performed. The logical values are the Boolean values TRUE and FALSE and the value NULL.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

variable_name datatype:= initial_value

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type.

```
DECLARE
   a integer :=10;
   b integer :=20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: '|| c);
   f :=70.0/3.0;
   dbms_output.put_line('Value of f: '|| f);
END;
/
```

When the above code is executed, it produces the following result −

```
Value of c: 30
Value of f: 23.333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form −

```
DECLARE
--Global variables
   num1 number :=95;
   num2 number :=85;
BEGIN
   dbms_output.put_line('Outer Variable num1: '|| num1);
   dbms_output.put_line('Outer Variable num2: '|| num2);
   DECLARE
--Local variables
      num1 number :=195;
      num2 number :=185;
BEGIN
      dbms_output.put_line('Inner Variable num1: '|| num1);
      dbms_output.put_line('Inner Variable num2: '|| num2);
END;
END;
/
```

When the above code is executed, it produces the following result −

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT into** statement of SQL to assign values to PL/SQL variables

Let us create a table named CUSTOMERS −

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR2(20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR(25),
   SALARY   DECIMAL(18,2),
   PRIMARY KEY (ID)
);
TableCreated
```

Now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1,'Ramesh',32,'Ahmedabad',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2,'Khilan',25,'Delhi',1500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3,'kaushik',23,'Kota',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4,'Chaitali',25,'Mumbai',6500.00);
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
   c_id customers.id%type :=1;
   c_name  customers.name%type;
   c_addr customers.address%type;
   c_sal  customers.salary%type;
BEGIN
   SELECT name, address, salary INTO c_name, c_addr, c_sal
   FROM customers
   WHERE id = c_id;
   dbms_output.put_line('Customer '||c_name ||' from '|| c_addr ||' earns '|| c_sal);
END;
/
```

When the above code is executed, it produces the following result −

Customer Ramesh from Ahmedabad earns 2000
PL/SQL procedure successfully completed.

Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example −

```
PI CONSTANT NUMBER :=3.141592654;
DECLARE
-- constant declaration
   pi constant number :=3.141592654;
   radius number(5,2);
   dia number(5,2);
   circumference number(7,2);
   area number (10,2);
BEGIN
   radius :=9.5;
   dia := radius *2;
   circumference :=2.0* pi * radius;
   area := pi * radius * radius;
   dbms_output.put_line('Radius: '|| radius);
   dbms_output.put_line('Diameter: '|| dia);
   dbms_output.put_line('Circumference: '|| circumference);
   dbms_output.put_line('Area: '|| area);
END;
/
```

output

Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.

Control structures

Conditional control

1. PL/SQL - IF-THEN Statement

It is the simplest form of the **IF** control statement, frequently used in decision-making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax

```
IF condition THEN
   Statements;
Endif;
```

Where *condition* is a Boolean or relational condition and S is a simple or compound statement. Following is an example of the IF-THEN statement −

```
IF (a <= 20) THEN
   c:= c+1;
END IF;
```

If the Boolean expression condition evaluates to true, then the block of code inside the **if statement** will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the **if statement** (after the closing end if) will be executed.

Example

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
a is less than 20
value of a is : 10

PL/SQL procedure successfully completed.
```

Example 2

Consider we have a table and few records in the table as we had created PL/SQL Variable Types

```
DECLARE
   c_id customers.id%type := 1;
   c_sal  customers.salary%type;
BEGIN
   SELECT  salary
   INTO  c_sal
   FROM customers
```

```
   WHERE id = c_id;
   IF (c_sal <= 2000) THEN
      UPDATE customers
      SET salary =  salary + 1000
         WHERE id = c_id;
      dbms_output.put_line ('Salary updated');
   END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Salary updated

PL/SQL procedure successfully completed.
```

1. PL/SQL - IF-THEN-ELSE Statement

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

Syntax

Syntax for the IF-THEN-ELSE statement is −

```
IF condition THEN
   S1;
ELSE
   S2;
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the **IF-THEN-ELSE statements**, when the test condition is TRUE, the statement *S1* is executed and *S2* is skipped; when the test condition is FALSE, then *S1* is bypassed and statement *S2* is executed. For example −

```
IF color = red THEN
  dbms_output.put_line('You have chosen a red car')
ELSE
  dbms_output.put_line('Please choose a color for your car');
END IF;
```

If the Boolean expression condition evaluates to true, then the **if-then block of code** will be executed otherwise the else block of code will be executed.

Example

```
DECLARE
   a number(3) := 100;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
```

```
    dbms_output.put_line('a is less than 20 ' );
  ELSE
    dbms_output.put_line('a is not less than 20 ' );
  END IF;
  dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
a is not less than 20
value of a is 100

PL/SQL procedure successfully completed.
```

2. PL/SQL - IF-THEN-ELSIF Statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement.

When using **IF-THEN-ELSIF** statements there are a few points that we have to consider

- It's ELSIF, not ELSEIF.
- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax

The syntax of an **IF-THEN-ELSIF** Statement in PL/SQL programming language is −

```
IF(boolean_expression 1)THEN
  S1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
  S2;  -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
  S3; -- Executes when the boolean expression 3 is true
ELSE
  S4; -- executes when the none of the above condition is true
END IF;
```

Example

```
DECLARE
  a number(3):=100;
BEGIN
  IF ( a =10) THEN
    dbms_output.put_line('Value of a is 10');
  ELSIF ( a =20) THEN
    dbms_output.put_line('Value of a is 20');
  ELSIF ( a =30) THEN
```

```
    dbms_output.put_line('Value of a is 30');
  ELSE
    dbms_output.put_line('None of the values is matching');
END IF;
  dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

None of the values is matching

PL/SQL procedure successfully completed.

PL/SQL - Nested IF-THEN-ELSE Statements

It is always legal in PL/SQL programming to nest the **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax

```
IF( boolean_expression 1)THEN
  -- executes when the boolean expression 1 is true
  IF(boolean_expression 2) THEN
    -- executes when the boolean expression 2 is true
    sequence-of-statements;
  END IF;
ELSE
  -- executes when the boolean expression 1 is not true
  else-statements;
END IF;
```

Example

```
DECLARE
  a number(3) := 100;
  b number(3) := 200;
BEGIN
  IF( a = 100 ) THEN
    IF( b = 200 ) THEN
    dbms_output.put_line('Value of a is 100 and b is 200' );
    END IF;
  END IF;
  dbms_output.put_line('Exact value of a is : ' || a );
  dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

PL/SQL procedure successfully completed.

Examples

Program to check whether student is pass or fail using if statement

```
DECLARE
  Marks number(2);
BEGIN
  Marks:=&marks;
 -- check the boolean condition using if statement
  IF( marks>=35 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('pass ' );
  END IF;
  dbms_output.put_line('marks is : ' || marks);
END;
```

Program to check whether student is pass or fail using if else statement

```
DECLARE
  Marks number(2);
BEGIN
  Marks:=&marks;
  IF( marks>=35 ) THEN
    dbms_output.put_line('pass ' || marks );
  ELSE
  dbms_output.put_line(' FAIL ' || marks);
END if;
END;
```

Program to find grades of a student by using if then elsif statement

```
DECLARE
  Marks number(2);
BEGIN
  Marks:=&marks;
  IF( marks>=70 and marks<=100) THEN
    dbms_output.put_line('DISTINCTION' );
  ELSIF( marks>=60 and marks<70) THEN
    dbms_output.put_line('FIRST CLASS' );
  ELSIF( marks>=50 and marks<60) THEN
    dbms_output.put_line('SECOND CLASS' );
  ELSIF( marks>=35 and marks<50) THEN
    dbms_output.put_line('THIRD CLASS' );
  ELSE
  dbms_output.put_line(' FAIL ' || marks);
END if;
END;
```

Iteration control

PL/SQL - Basic Loop Statement

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax

```
Declare
initialization
Begin
LOOP
  Sequence of statements;
exit or exit when;
END LOOP;
```

Here, the sequence of statement(s) may be a single statement or a block of statements. An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

Example

```
DECLARE
   x number :=10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x +10;
      IF x >50 THEN
exit;
END IF;
END LOOP;
   dbms_output.put_line('After Exit x is: '|| x);
END;
/
```

Output

```
10
20
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

You can use the **EXIT WHEN** statement instead of the **EXIT** statement −

```
DECLARE
   x number :=10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x +10;
exit WHEN x >50;
END LOOP;
   dbms_output.put_line('After Exit x is: '|| x);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

PL/SQL - FOR LOOP Statement

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

```
FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

Example

```
DECLARE
   a number(2);
BEGIN
   FOR a in10..20 LOOP

      dbms_output.put_line('value of a: '|| a);
END LOOP;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this −

```
DECLARE
   a number(2);
BEGIN
```

```
   FOR a IN REVERSE 10..20 LOOP
      dbms_output.put_line('value of a: '|| a);
END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

PL/SQL - WHILE LOOP Statement

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example

```
DECLARE
   a number(2):=10;
BEGIN
   WHILE a <20 LOOP
      dbms_output.put_line('value of a: '|| a);
      a := a +1;
END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

```
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

Sequential control

PL/SQL - GOTO Statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

Syntax

The syntax for a GOTO statement in PL/SQL is as follows −

```
GOTO label;
..
<< label >>
statement;
```

Example

```
DECLARE
   a number(2):=10;
BEGIN
<<loopstart>>
   WHILE a <20 LOOP
   dbms_output.put_line ('value of a: '|| a);
      a := a +1;
      IF a =15 THEN
         a := a +1;
         GOTO loopstart;
END IF;
END LOOP;
END;
/
```

Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

PL/SQL procedure successfully completed.

The NULL Statement

Usually we write a statement in a program, we want it to do something. There are cases, however, when we want to tell PL/SQL to do absolutely nothing, and that is where the NULL statement comes in existence. The NULL statement has the following format:

NULL;

The NULL statement is simply the reserved word NULL followed by a semicolon (;) to indicate that this is a statement and not a NULL value. The NULL statement does nothing except pass control to the next executable statement.

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
GOTO end_loop;
    END IF;
<<end_loop>>
NULL;
  END LOOP;
END;
/
```

Exception Handling

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions –

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. The default exception will be handled using ***WHEN others THEN*** −

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
  WHEN exception1 THEN
    exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

```
DECLARE
  c_id customers.id%type :=8;
  c_name customerS.Name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: '|| c_addr);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION block**.

Predefined exceptions

1. **NO_DATA_FOUND**: It is raised WHEN a SELECT INTO statement returns *no* rows. For eg:

```
DECLARE
  tempvarchar(20);
BEGIN
  SELECTg_id intotempfromgeeks whereg_name='rahul';
exception
  WHENno_data_found THEN
    dbms_output.put_line('ERROR');
    dbms_output.put_line('there is no name as');
    dbms_output.put_line('rahul in geeks table');
end;
```

Output:

ERROR

there is no name as rahul in geeks table

2. **TOO_MANY_ROWS**:It is raised WHEN a SELECT INTO statement returns *more* than one row.

```
DECLARE
  tempvarchar(20);
BEGIN
  SELECTg_name intotempfromgeeks;
  dbms_output.put_line(temp);
EXCEPTION
  WHENtoo_many_rows THEN
    dbms_output.put_line('error trying to SELECT too many rows');
end;
```

error trying to SELECT too many rows

3. **VALUE_ERROR**:This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

```
DECLARE
  tempnumber;
BEGIN
  SELECTg_name  intotempfromgeeks whereg_name='Suraj';
  dbms_output.put_line('the g_name is '||temp);
EXCEPTION
  WHENvalue_error THEN
  dbms_output.put_line('Error');
  dbms_output.put_line('Change data type of temp to varchar(20)');

END;
```

Output:

Error

Change data type of temp to varchar(20)

4. **ZERO_DIVIDE** = raises exception WHEN dividing with zero.

```
DECLARE
  a int:=10;
  b int:=0;
```

```
   answer int;
BEGIN
   answer:=a/b;
   dbms_output.put_line('the result after division is'||answer);
exception
   WHENzero_divide THEN
      dbms_output.put_line('dividing by zero please check the values again');
      dbms_output.put_line('the value of a is '||a);
      dbms_output.put_line('the value of b is '||b);
END;
```

Output:

dividing by zero please check the values again

the value of a is 10

the value of b is 0

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
  my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
   c_id customers.id%type :=&cc_id;
   c_name customerS.Name%type;
   c_addr customers.address%type;
-- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <=0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: '|| c_addr);
END IF;

EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Enter value for cc_id: -6
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

PL/SQL - Cursors

A cursor is a temporary work area created in system memory when a SQL statement is executed. A cursor is a set of rows together with a pointer that identifies a current row. It is a database object to retrieve data from a result set one row at a time. It is useful when we want to manipulate the record of a table in a singleton method, in other words one row at a time. In other words, a cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name**.

Example

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

Output

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME| AGE | ADDRESS | SALARY|
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

```
DECLARE
   c_id customers.id%type;
   c_name customerS.Noame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id ||' '|| c_name ||' '|| c_addr);
END LOOP;
   CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result−

1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.

PL/SQL Parameterized Cursor

PL/SQL Parameterized cursor pass the parameters into a cursor and use them in to query.

length. Parameterized cursors are also saying static cursors that can passed parameter value when cursor are opened.

The %ROWTYPE attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.

Following example shows the parameterized cursor on emp_information table,

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45k
2	marks jems	Program Developer	38k
3	Saulin	Program Developer	34k
4	Zenia Sroll	Web Developer	42k

Example Cursor display employee information from emp_information table whose emp_no four (4) :

```
SQL>DECLARE
cursorc(no number)isselect*from emp_information
where emp_no =no;
  tmp emp_information%rowtype;
BEGIN
OPENc(4);
FOR tmp INc(4)LOOP
  dbms_output.put_line('EMP_No:    '||tmp.emp_no);
  dbms_output.put_line('EMP_Name:  '||tmp.emp_name);
  dbms_output.put_line('EMP_Dept:  '||tmp.emp_dept);
```

```
  dbms_output.put_line('EMP_Salary:'||tmp.emp_salary);
ENDLoop;
CLOSEc;
END;
/
```

Result

```
EMP_No:    4
EMP_Name:  Zenia Sroll
EMP_Dept:  Web Developer
EMP_Salary:  42k
PL/SQL procedure successfully completed.
```

Procedure in PL/SQL

A Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure has its own unique name by which it can be referred.

Advantages

To help you build powerful database applications, stored procedures provide several advantages including better performance, higher productivity, ease of use, and increased scalability.

Performance

Stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. Executable code is automatically cached and shared among users. This lowers memory requirements and invocation overhead.

By grouping SQL statements, a stored procedure allows them to be executed with a single call. This minimizes the use of slow networks, reduces network traffic, and improves round-trip response time. OLTP applications, in particular, benefit because result set processing eliminates network bottlenecks.

Productivity and Ease of Use

By designing applications around a common set of stored procedures, you can avoid redundant coding and increase your productivity. Moreover, stored procedures let you extend the functionality of the RDBMS. For example, stored functions called from SQL statements enhance the power of SQL.

You can use the Java integrated development environment (IDE) of your choice to create stored procedures. Then, you can deploy them on any tier of the network architecture. Moreover, they can be called by standard Java interfaces such as JDBC, CORBA, and EJB and by programmatic interfaces and development tools such as SQLJ, the OCI, Pro*C/C++, and JDeveloper.

Scalability

Stored procedures increase scalability by isolating application processing on the server. In addition, automatic dependency tracking for stored procedures aids the development of scalable applications.

Once it is validated, a stored procedure can be used with confidence in any number of applications. If its definition changes, only the procedure is affected, not the applications that call it. This simplifies maintenance and enhancement. Also, maintaining a procedure on the server is easier than maintaining copies on various client machines.

Interoperability

Within the RDBMS, Java conforms fully to the *Java Language Specification* and furnishes all the advantages of a general-purpose, object-oriented programming language. Also, like PL/SQL, Java provides full access to Oracle data, so any procedure written in PL/SQL can be written in Java.

The RDBMS allows a high degree of interoperability between Java and PL/SQL. Java applications can call PL/SQL stored procedures using an embedded JDBC driver. Conversely, PL/SQL applications can call Java stored procedures directly.

Security

You can restrict access to data by allowing users to manipulate the data only through stored procedures that execute with their definer's privileges. For example, you can allow access to a procedure that updates a database table, but deny access to the table itself.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [,...])]
IS
BEGIN
< procedure_body >
```

```
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

Output

Procedure created.

Executing a Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
   greetings;
END;
/
```

The above call will display −

Hello World

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
  a number;
  b number;
```

```
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
END IF;
END;
BEGIN
   a:=23;
   b:=45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:=23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Output

Square of (23): 529

PL/SQL procedure successfully completed.

Functions

Advantages:

1. We can make a single call to the database to run a block of statements thus it improves the performance against running SQL multiple times. This will reduce the number of calls between the database and the application.

2. We can divide the overall work into small modules which becomes quite manageable also enhancing the readability of the code.
3. It promotes reusability.
4. It is secure since the code stays inside the database thus hiding internal database details from the application(user). The user only makes a call to the PL/SQL functions. Hence security and data hiding is ensured.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
IS
BEGIN
< function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
```

```
RETURN number IS
  total number(2):=0;
BEGIN
  SELECT count(*)into total
  FROM customers;

  RETURN total;
END;
/
```

Output

Function created.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block –

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: '|| c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Total no. of Customers: 6

PL/SQL procedure successfully completed.

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
  a number;
  b number;
  c number;
```

```
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
     z:= x;
  ELSE
     Z:= y;
END IF;
  RETURN z;
END;
BEGIN
  a:=23;
  b:=45;
  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): '|| c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

Triggers:

A MySQL **trigger** is a stored program (with queries) which is executed automatically to respond to a specific event such as insertion, updation or deletion occurring in a table.

Uses for triggers:

- Validate input data
- Generate a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes
- Query from other files for cross-referencing purposes
- Access system functions
- Replicate data to different files to achieve data consistency

Creating Triggers

The syntax for creating a trigger is −

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR]| UPDATE [OR]| DELETE}
[OF col_name]
ON table_name
```

```
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example : **CUSTOMERS table**

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID >0)
```

```
DECLARE
  sal_diff number;
BEGIN
  sal_diff := NEW.salary  - OLD.salary;
  dbms_output.put_line('Old salary: '|| OLD.salary);
  dbms_output.put_line('New salary: '|| NEW.salary);
  dbms_output.put_line('Salary difference: '|| sal_diff);
END;
/
```

Invoking a Trigger

Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7,'Kriti',22,'HP',7500.00);
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null.The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary +500
WHERE id =2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

DROP TRIGGER - In MySQL Trigger can also be drop. When Trigger drops, then it is removed from the database.

Syntax**Drop TriggerTrigger_name;**

ExampleDrop Trigger student_update;

UNIT-6 :Introduction to Nosql database

NoSQL Databases

We know that MongoDB is a NoSQL Database, so it is very necessary to know about NoSQL Database to understand MongoDB throughly.

What is NoSQL Database

Databases can be divided in 3 types:

1. RDBMS (Relational Database Management System)
2. OLAP (Online Analytical Processing)
3. NoSQL (recently developed database)

NoSQL Database

NoSQL Database is used to refer a non-SQL or non relational database.

It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases. NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

History behind the creation of NoSQL Databases

In the early 1970, Flat File Systems are used. Data were stored in flat files and the biggest problems with flat files are each company implement their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.

Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.

Advantages of NoSQL

- It supports query language.
- It provides fast performance.
- It provides horizontal scalability.

MongoDB advantages over RDBMS

In recent days, MongoDB is a new and popularly used database. It is a document based, non relational database provider.

Although it is 100 times faster than the traditional database but it is early to say that it will broadly replace the traditional RDBMS. But it may be very useful in term to gain performance and scalability.

A Relational database has a typical schema design that shows number of tables and the relationship between these tables, while in MongoDB there is no concept of relationship.

MongoDB Advantages

- **MongoDB is schema less**. It is a document database in which one collection holds different documents.
- There may be **difference between number of fields, content and size of the document** from one to other.
- **Structure of a single object is clear** in MongoDB.
- There are **no complex joins** in MongoDB.
- MongoDB provides the **facility of deep query** because it supports a powerful dynamic query on documents.
- It is very **easy to scale**.
- It **uses internal memory for storing working sets** and this is the reason of its fast access.

Distinctive features of MongoDB

- Easy to use
- Light Weight
- Extremely faster than RDBMS

Where MongoDB should be used

- Big and complex data
- Mobile and social infrastructure
- Content management and delivery
- User data management
- Data hub

Performance analysis of MongoDB and RDBMS

- In relational database (RDBMS) tables are using as storing elements, while in MongoDB collection is used.
- In the RDBMS, we have multiple schema and in each schema we create tables to store data while, MongoDB is a document oriented database in which data is written in BSON format which is a JSON like format.
- MongoDB is almost 100 times faster than traditional database systems.

How to install MongoDB on Windows

Firstly you will have to download the latest release of MongoDB:

How to download MongoDB

You can download an appropriate version of MongoDB which your system supports, from the link **"http://www.mongodb.org/downloads"** to install the MongoDB on Windows. You should choose correct version of MongoDB acording to your computer's Window. If you are not sure what Window version are you using, open your command prompt and execute this command:

1. C:\ wmic os get osarchitecture

```
OSArchitecture
64 bit
C:\ >
```

Note: MongoDB does not support Window XP.

MongoDB for Windows Server 2008 R2 edition

This version of MongoDB runs only on Window Server 2008 R2, Window7 64 bit, and the newer version of windows. You can't operate it on older version of windows.

MongoDb for 64 bit Windows

This version of MongoDB runs only on newer version of Windows contains 64 bit operating system.

for example: Window Server 2008 R2, Window 7 64 bit etc.

MongoDb for 32 bit Windows

This version of MongoDB runs on only 32 bit windows. 32 bit version of MongoDB is generally used in testing and development purposes because it supports databases smaller than 2 GB.

How to install the downloaded file

In Window explorer, locate the downloaded MongoDB msi file, double click on that file and follow the instructions appears on the screen. These instructions will guide you to complete the installation process.

Note: If you want to move the MongoDB folder from default position to another position, it is necessary to issue the move command as an administrator. let us take an example to move the folder to C : \mongodb:

1. **Select** Start Menu > All Programs > Accessories

You can install MongoDB in any folder because it is self contained and does not have any other system dependency.

How to set up the MongoDB environment

A data directory is required in MongoDB to store all the information. Its by default data directory path is \data\db. you can create this folder by command prompt.

1. md\data\db

For example:

If you want to start MongoDB, run mongod.exe

You can do it from command prompt.

1. C:\Program Files\MongoDB\bin\mongod.exe

This will start the mongoDB database process. If you get a message "waiting for connection" in the console output, it indicates that the mongodb.exe process is running successfully.

For example:

When you connect to the MongoDB through the mongo.exe shell, you should follow these steps:

1. Open another command prompt.
2. At the time of connecting, specify the data directory if necessary.

> Note: If you use the default data directory while MongoDB installation, there is no need to specify the data directory.

For example:

1. C:\mongodb\bin\mongo.exe

If you use the different data directory while MongoDB installation, specify the directory when connecting.

For example:

1. C:\mongodb\bin\mongod.exe-- dbpath d:\test\mongodb\data

If you have spaces in your path, enclose the entire path in double space.

For example:

```
C:\mongodb\bin\mongod.exe-- dbpath  "d: \ test\mongodb\data"
```

How to configure directory and files

First to create a configuration file and a directory path for MongoDB log output after that create a specific directory for MongoDB log files.

MongoDB Database commands

The MongoDB database commands are used to create, modify, and update the database.

#1. db.adminCommand(cmd)

The admin command method runs against the admin database to run specified database commands by providing a helper.

Command: Either the argument is specified in the document form or a string form. If the command is defined as a string, it cannot include any argument.

Example:

Creating a user named JavaTpoint with the dbOwner role on the admin database.

```
db.adminCommand(
  {
    createUser: "JavaTpoint",
```

```
4.      pwd: passwordPrompt(),
5.      roles: [
6.        { role: "dbOwner", db: "admin" }
7.      ]
8.    }
9.  )
```

Output:

```
mongodb@ubuntu:~/mongodb-linux-x86_64-2.6.0$ bin/mongo
MongoDB shell version: 2.6.0
connecting to: test
> db.adminCommand( { getCmdLineOpts: 1 } )
{
        "argv" : [
                "bin/mongod",
                "--dbpath",
                "data"
        ],
        "parsed" : {
                "storage" : {
                        "dbPath" : "data"
                }
        },
        "ok" : 1
}
>
```

#2. db.aggregate()

The aggregate method initialize a specific diagnostic or admin pipeline, which does not require anu underlying collection.

Syntax:

1. db.aggregate([**<pipeline>**], { **<options>** })

The pipeline parameter does not require any underlying collection and always starts with a compatible stage, such as $currentOp or $listLocalSessions. It is an array of stages that will be executed.

Example:

The following example runs a pipeline with two stages. The first is the $currentOp operation and the second will filters the results.

```
use admin
db.aggregate( [ {
   $currentOp : { allUsers: true, idleConnections: true } },
{
   $match : { shard: "shardDemo" }
   }
] )
```

Output:

```
> db.writers.find().pretty()
{
        "_id" : ObjectId("5d16f6a1e198c897a4105d0d"),
        "title" : "Networking",
        "description" : "Networking Essentials",
        "author" : "Gaurav Mandes"
}
{
        "_id" : ObjectId("5d16f6a1e198c897a4105d0e"),
        "title" : "Game Engineering",
        "description" : "Game Engineering and Development",
        "author" : "Gaurav Mandes"
}
{
        "_id" : ObjectId("5d16f6a1e198c897a4105d0f"),
        "title" : "PHP",
        "description" : "PHP as a General-Purpose",
        "author" : "Gautam"
}
>
```

#3. db.cloneDatabase("hostname")

The clonedatabase method copies the specified database to the current database and assumes that the database at the remote location has the same name as the current database.

The hostname parameter contains the hostname of the database that we want to copy.

Example:

db.cloneDatabase("customers")

Output:

#4. db.commandHelp(command)

We have the help option for the specified database command using the commandHelp method. The command parameter contains the name of a database command.

```
mongodb@ubuntu:~/mongodb-linux-x86_64-2.6.0$ bin/mongo
MongoDB shell version: 2.6.0
connecting to: test
> help
    db.help()                    help on db methods
    db.mycoll.help()             help on collection methods
    sh.help()                    sharding helpers
    rs.help()                    replica set helpers
    help admin                   administrative help
    help connect                 connecting to a db help
    help keys                    key shortcuts
    help misc                    misc things to know
    help mr                      mapreduce

    show dbs                     show database names
    show collections             show collections in current database
    show users                   show users in current database
    show profile                 show most recent system.profile
                                 entries with time >= 1ms
    show logs                    show the accessible logger names
    show log [name]              prints out the last segment of log
                                 in memory, 'global' is default
    use <db_name>                set current database
    db.foo.find()                list objects in collection foo
    db.foo.find( { a : 1 } )     list objects in foo where a == 1
    it                           result of the last line evaluated;
                                 use to further iterate
    DBQuery.shellBatchSize = x   set default number of items to
                                 display on shell
    exit                         quit the mongo shell
>
```

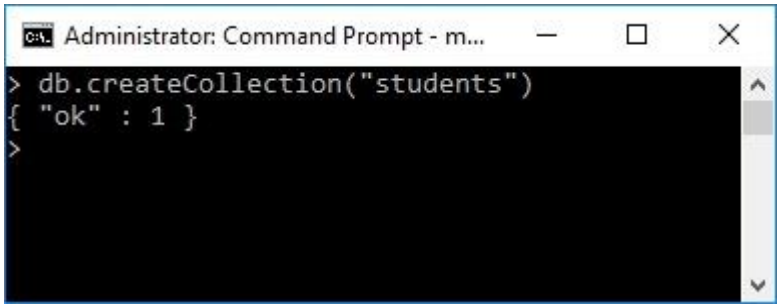
#5. db.createCollection(name, options)

A new collection or view will be created using this method. The createCollection method is used primarily for creating new collections that use specific options when the collection is first referenced in a command.

For example - we will create a javaTpoint collection with a JSON Schema validator:

```
1. db.createCollection( "student", {
2.    validator: { $jsonSchema: {
3.       bsonType: "object",
4.       required: [ "phone" ],
```

```
5.        properties: {
6.           phone: {
7.              bsonType: "string",
8.              description: "must be a string and is required"
9.           },
10.          email: {
11.             bsonType : "string",
12.             pattern: "@mongodb\.com$",
13.             description: "must be a string and match the regular expression pattern"
14.          },
15.          status: {
16.             enum: [ "Unknown", "Incomplete" ],
17.             description: "can only be one of the enum values"
18.          }
19.       }
20.    } }
21. } )
```



#6. db.createView()

When we applying the specified aggregation pipeline to the collection, the createView method create a new view for the collection. The method can be computed during the read operations and acts as read-only operations. The views can be created in the same database of the source collection to executes read operations as a part of the underlying aggregation pipeline.

Syntax:

```
1. db.createView(<view>, <source>, <pipeline>, <options>)
```

The following example creates a StudentFeedback view with the _id, student.management, and department fields:

```
db.createView(
   "StudentFeedback",
   "survey",
   [ { $project: { "management": "$Student.management", department: 1 } } ]
)
```

Output:

#7. db.dropDatabase(<writeConcern>)

The drop method removes the specified database and the associated data files.

For example -

we use <database> operation to switch the current database to the temporary database. We use the db.dropDatabase() method to drops the temporary database

```
use temp
db.dropDatabase()
```

#8. db.getLogComponents()

The getLog method returns the current stiltedly settings. The method determines the amount of Log Messages produced by MongoDB for each log message component.

Example:

```
{
    "verbosity" : 0,
    "accessControl" : {
        "verbosity" : -1
    },
    "command" : {
        "verbosity" : -1
    },
    "control" : {
        "verbosity" : -1
    },
    "geo" : {
        "verbosity" : -1
    },
    "index" : {
        "verbosity" : -1
    },
    "network" : {
        "verbosity" : -1
    },
    "query" : {
        "verbosity" : 2
    },
    "replication" : {
        "verbosity" : -1,
        "election" : {
            "verbosity" : -1
        },
        "heartbeats" : {
            "verbosity" : -1
        },
        "initialSync" : {
            "verbosity" : -1
        },
        "rollback" : {
            "verbosity" : -1
        }
```

```
38.    },
39.    "sharding" : {
40.       "verbosity" : -1
41.    },
42.    "storage" : {
43.       "verbosity" : 2,
44.       "recovery" : {
45.          "verbosity" : -1
46.       },
47.       "journal" : {
48.          "verbosity" : -1
49.       }
50.    },
51.    "write" : {
52.       "verbosity" : -1
53.    }
54. }
```

Cassandra vs MongoDB

Cassandra and MongoDB both are types of NoSQL databases. Cassandra is a distributed database system designed to handle large amount of data and known for its high scalability and high performance. While, MongoDB is document oriented database which also provides high scalability, high performance and automatic scaling.

In terms of simplicity, databases can be divided in two types:

- Development simplicity
- Operational simplicity

While MongoDB is known for an easy out-of-the-box experience, Cassandra is known for easy to manage at scale.

Following is a list of important differences between them:

Index	Cassandra	Mongodb
1)	Cassandra is high performance distributed database system.	MongoDB is cross-platform document- database system.
2)	Cassandra is written in Java.	MongoDB is written in C++.
3)	Cassandra stores data in tabular form like SQL format.	MongoDB stores data in JSON format.
4)	Cassandra is got license by Apache.	MongoDB is got license by AGPL and d Apache.
5)	Cassandra is mainly designed to handle large amounts of data across many commodity servers.	MongoDB is designed to deal with J documents and access applications ea faster.
6)	Cassandra provides high availability with no single point of failure.	MongoDB is easy to administer in the failure.

Key Points of Apache Cassandra

- Cassandra is highly scalable, high performance, consistent and fault-tolerant database system. Cassandra is a column-oriented database.
- Cassandra provides easy data distribution.
- Cassandra supports ACID properties i.e. Atomicity, Consistency, Isolation, and Durability.
- Cassandra follows the distribution design of Amazon?s dynamo and its data model design is based on Google's Bigtable.

- Cassandra was initially created at Facebook for inbox search and now it is being used by some of the biggest companies like Facebook, Twitter, ebay, Netflix, Cisco, Rackspace etc.

Key Points of MongoDB

- MongoDB is well suited for Bigdata and mobile & social infrastructure.
- MongoDB provides Replication, High availability and Auto-sharding.
- MongoDB is used by companies like Foursquare, Intuit, Shutterfly, SourceForge, The New York Times, Lexis Nexis Orange Digital etc.

MongoDB Tutorial

MongoDB tutorial provides basic and advanced concepts of SQL. Our MongoDB tutorial is designed for beginners and professionals.

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.

Our MongoDB tutorial includes all topics of MongoDB database such as insert documents, update documents, delete documents, query documents, projection, sort() and limit() methods, create a collection, drop collection, etc. There are also given MongoDB interview questions to help you better understand the MongoDB database.

What is MongoDB

MongoDB

is an open-source document database that provides high performance, high availability, and automatic scaling.

In simple words, you can say that - Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.

MongoDB is available under General Public license for free, and it is also available under Commercial license from the manufacturer.

The manufacturing company 10gen has defined MongoDB as:

"MongoDB is a scalable, open source, high performance, document-oriented database." - 10gen

MongoDB was designed to work with commodity servers. Now it is used by the company of all sizes, across all industry.

History of MongoDB

The initial development of MongoDB began in 2007 when the company was building a platform as a service similar to window azure.

> Window azure is a cloud computing platform and infrastructure, created by Microsoft, to build, deploy and manage applications and service through a global network.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc. It was initially developed as a PAAS (Platform as a Service). Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

MongoDB2.4.9 was the latest and stable version which was released on January 10, 2014.

Purpose of building MongoDB

It may be a very genuine question that - "what was the need of MongoDB although there were many databases in action?"

There is a simple answer:

All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.

The primary purpose of building MongoDB is:

- Scalability
- Performance
- High Availability
- Scaling from single server deployments to large, complex multi-site architectures.
- Key points of MongoDB
- Develop Faster
- Deploy Easier
- Scale Bigger

First of all, we should know what is document oriented database?

Example of document oriented database

MongoDB is a document oriented database. It is a key feature of MongoDB. It offers a document oriented storage. It is very simple you can program it easily.

MongoDB stores data as documents, so it is known as document-oriented database.

1. FirstName = "John",

2. Address = "Detroit",
3. Spouse = [{**Name**: "Angela"}].

4. FirstName ="John",
5. Address = "Wick"

There are two different documents (separated by ".").

Storing data in this manner is called as document-oriented database.

Mongo DB falls into a class of databases that calls Document Oriented Databases. There is also a broad category of database known as No SQL Databases

.

Features of MongoDB

These are some important features of MongoDB:

1. Support ad hoc queries

In MongoDB, you can search by field, range query and it also supports regular expression searches.

2. Indexing

You can index any field in a document.

3. Replication

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

4. Duplication of data

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

5. Load balancing

It has an automatic load balancing configuration because of data placed in shards.

6. Supports map reduce and aggregation tools.

7. Uses JavaScript

instead of Procedures.

8. It is a schema-less database written in C++

.

9. Provides high performance.

10. Stores files of any size easily without complicating your stack.

11. Easy to administer in the case of failures.

12. It also supports:

JSON data model with dynamic schemas

Auto-sharding for horizontal scalability

Built in replication for high availability

Now a day many companies using MongoDB to create new types of applications, improve performance and availability.

Prerequisite

Before learning MongoDB, you must have the basic knowledge of SQL and OOPs.

Audience

Our MongoDB tutorial is designed to help beginners and professionals.

Problem

We assure that you will not find any problem in this MongoDB tutorial. But if there is any mistake, please post the problem in contact form.

What is a NoSQL database?

When people use the term “NoSQL database,” they typically use it to refer to any non-relational database. Some say the term “NoSQL” stands for “non SQL” while others say it stands for “not only SQL.” Either way, most agree that NoSQL databases are databases that store data in a format other than relational tables.

Brief history of NoSQL databases

NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased. Gone were the days of needing to create a complex, difficult-to-manage data model in order to avoid data duplication. Developers (rather than storage) were becoming the primary cost of software development, so NoSQL databases optimized for developer productivity.

As storage costs rapidly decreased, the amount of data that applications needed to store and query increased. This data came in all shapes and sizes — structured, semi-structured, and polymorphic — and defining the schema in advance became nearly impossible. NoSQL databases allow developers to store huge amounts of unstructured data, giving them a lot of flexibility.

Additionally, the Agile Manifesto was rising in popularity, and software engineers were rethinking the way they developed software. They were recognizing the need to rapidly adapt to changing requirements. They needed the ability to iterate quickly and make changes throughout their software stack — all the way down to the database. NoSQL databases gave them this flexibility.

Cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. They wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up, and to intelligently geo-place their data. Some NoSQL databases like MongoDB provide these capabilities.

NoSQL database features

Each NoSQL database has its own unique features. At a high level, many NoSQL databases have the following features:

- Flexible schemas
- Horizontal scaling

- Fast queries due to the data model
- Ease of use for developers

Check out What are the Benefits of NoSQL Databases? to learn more about each of the features listed above.

Types of NoSQL databases

Over time, four major types of NoSQL databases emerged: document databases, key-value databases, wide-column stores, and graph databases.

- Document databases store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
- Key-value databases are a simpler type of database where each item contains keys and values.
- Wide-column stores store data in tables, rows, and dynamic columns.
- Graph databases store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

To learn more, visit Understanding the Different Types of NoSQL Databases.

Difference between RDBMS and NoSQL databases

While a variety of differences exist between relational database management systems (RDBMS) and NoSQL databases, one of the key differences is the way the data is modeled in the database. In this section, we'll work through an example of modeling the same data in a relational database and a NoSQL database. Then, we'll highlight some of the other key differences between relational databases and NoSQL databases.

RDBMS vs NoSQL: Data Modeling Example

Let's consider an example of storing information about a user and their hobbies. We need to store a user's first name, last name, cell phone number, city, and hobbies.

In a relational database, we'd likely create two tables: one for Users and one for Hobbies.

Users

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

In order to retrieve all of the information about a user and their hobbies, information from the Users table and Hobbies table will need to be joined together.

The data model we design for a NoSQL database will depend on the type of NoSQL database we choose. Let's consider how to store the same information about a user and their hobbies in a document database like MongoDB.

```
{
   "_id": 1,
   "first_name": "Leslie",
   "last_name": "Yepp",
   "cell": "8125552344",
   "city": "Pawnee",
   "hobbies": ["scrapbooking", "eating waffles", "working"]
}
```

In order to retrieve all of the information about a user and their hobbies, a single document can be retrieved from the database. No joins are required, resulting in faster queries.

To see a more detailed version of this data modeling example, read Mapping Terms and Concepts from SQL to MongoDB.

Other differences between RDBMS and relational databases

While the example above highlights the differences in data models between relational databases and NoSQL databases, many other important differences exist, including:

- Flexibility of the schema
- Scaling technique
- Support for transactions
- Reliance on data to object mapping

To learn more about the differences between relational databases and NoSQL databases, visit NoSQL vs SQL Databases.

Why NoSQL?

NoSQL databases are used in nearly every industry. Use cases range from the highly critical (e.g., storing financial data and healthcare records) to the more fun and frivolous (e.g., storing IoT readings from a smart kitty litter box).

In the following sections, we'll explore when you should choose to use a NoSQL database and common misconceptions about NoSQL databases.

When should NoSQL be used?

When deciding which database to use, decision-makers typically find one or more of the following factors lead them to selecting a NoSQL database:

- Fast-paced Agile development
- Storage of structured and semi-structured data
- Huge volumes of data
- Requirements for scale-out architecture
- Modern application paradigms like microservices and real-time streaming

See When to Use NoSQL Databases and Exploring NoSQL Database Examples for more detailed information on the reasons listed above.

NoSQL database misconceptions

Over the years, many misconceptions about NoSQL databases have spread throughout the developer community. In this section, we'll discuss two of the most common misconceptions:

- Relationship data is best suited for relational databases.
- NoSQL databases don't support ACID transactions.

To learn more about common misconceptions, read Everything You Know About MongoDB is Wrong.

Misconception: relationship data is best suited for relational databases

A common misconception is that NoSQL databases or non-relational databases don’t store relationship data well. NoSQL databases can store relationship data — they just store it differently than relational databases do.

In fact, when compared with relational databases, many find modeling relationship data in NoSQL databases to be easier than in relational databases, because related data doesn’t have to be split between tables. NoSQL data models allow related data to be nested within a single data structure.

Misconception: NoSQL databases don't support ACID transactions

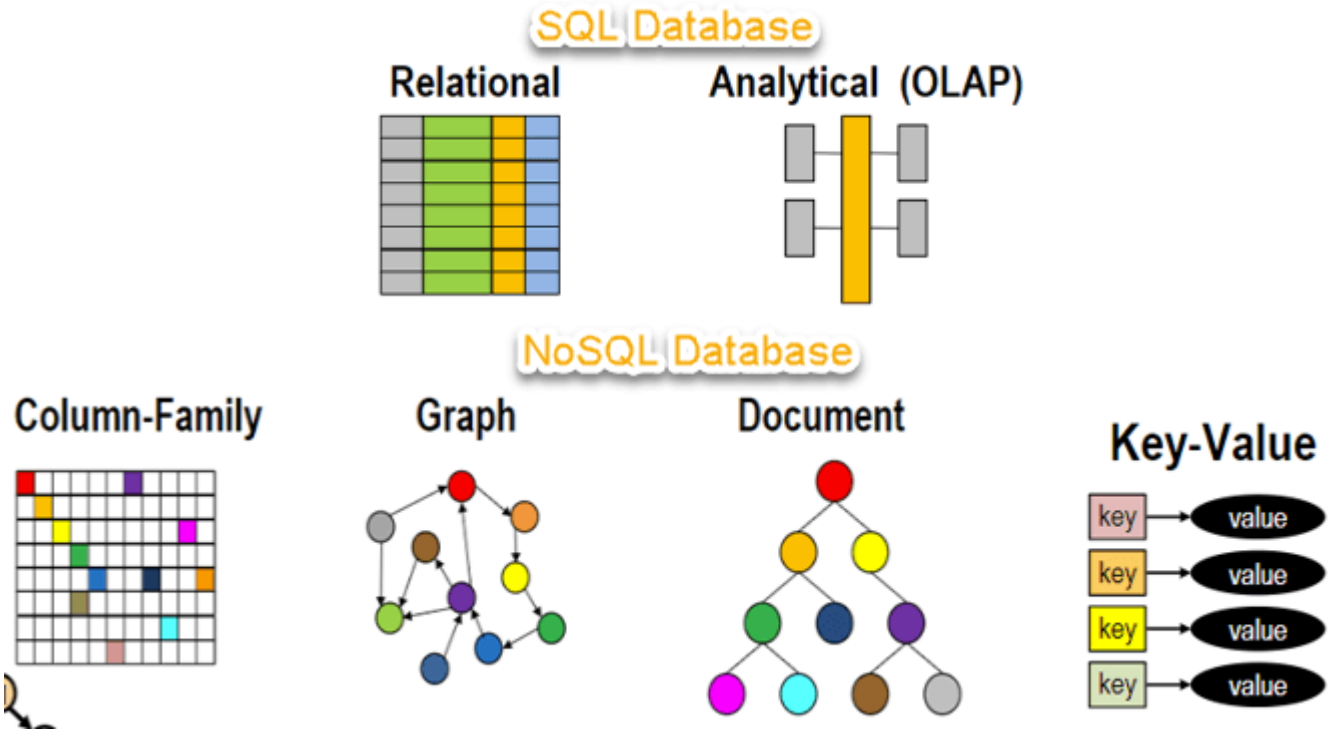
Another common misconception is that NoSQL databases don't support ACID transactions. Some NoSQL databases like MongoDB do, in fact, support ACID transactions.

Note that the way data is modeled in NoSQL databases can eliminate the need for multi-record transactions in many use cases. Consider the earlier example where we stored information about a user and their hobbies in both a relational database and a document database. In order to ensure information about a user and their hobbies was updated together in a relational database, we'd need to use a transaction to update records in two tables. In order to do the same in a document database, we could update a single document — no multi-record transaction required.\

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.
NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:



How to

In this NoSQL tutorial for beginners, you will learn NoSQL basics like:

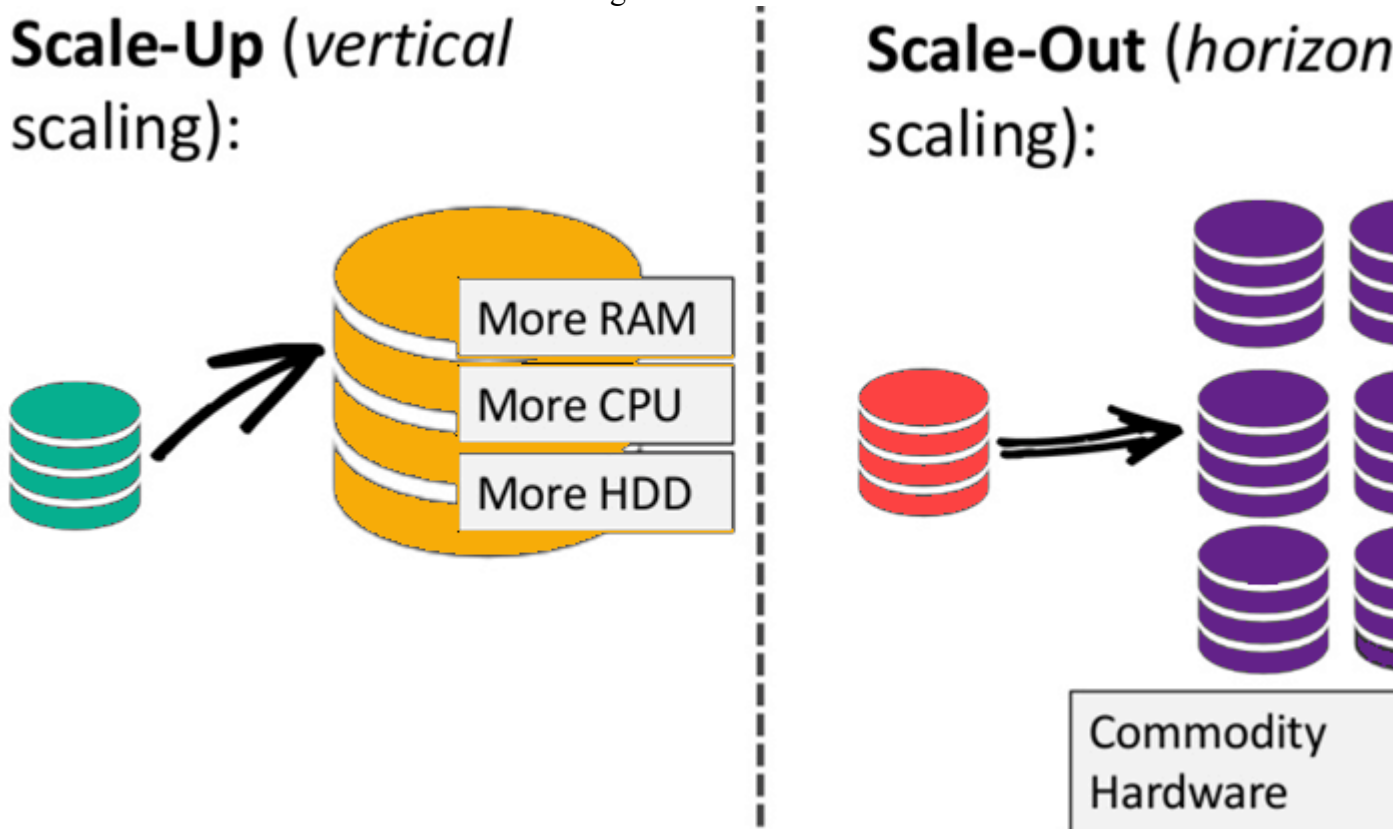
- Disadvantages of NoSQL

Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.

The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as “scaling out.”



NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released

- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

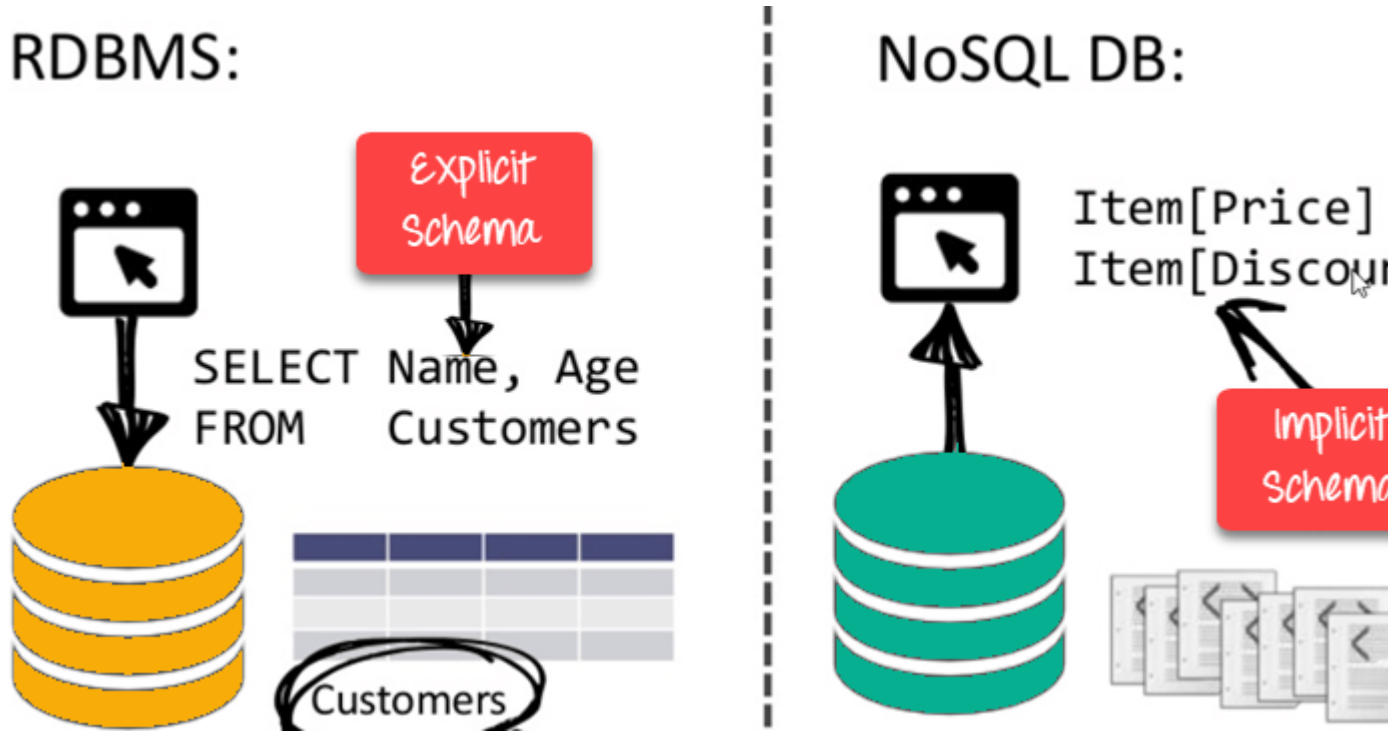
Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain



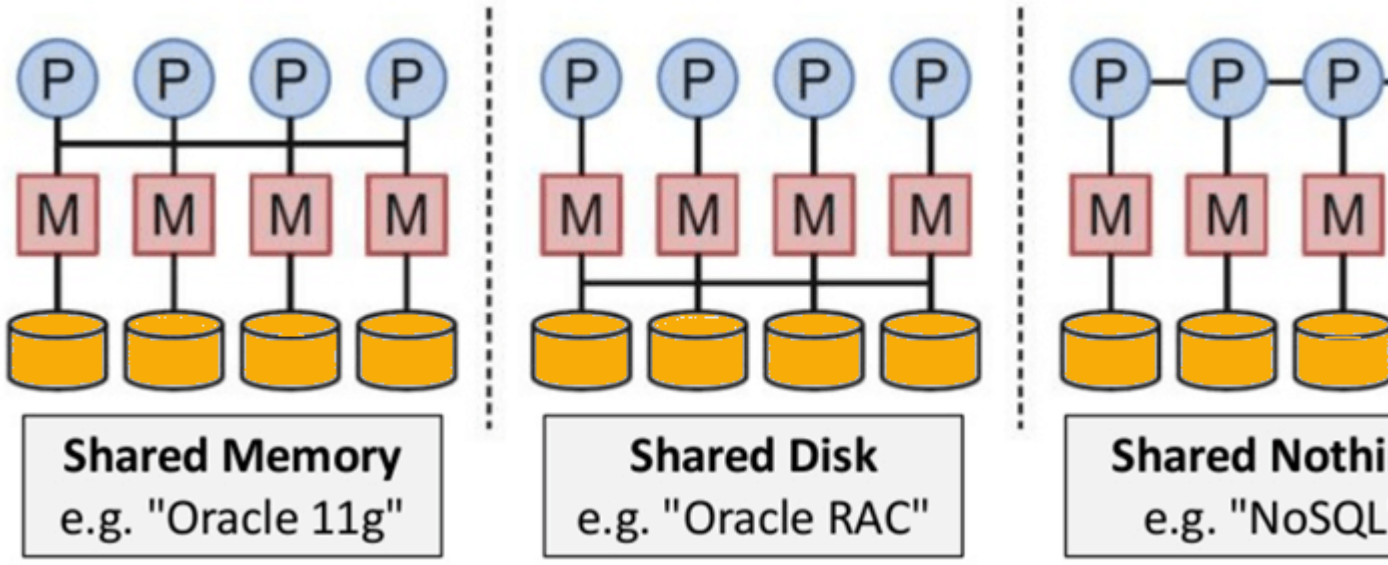
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

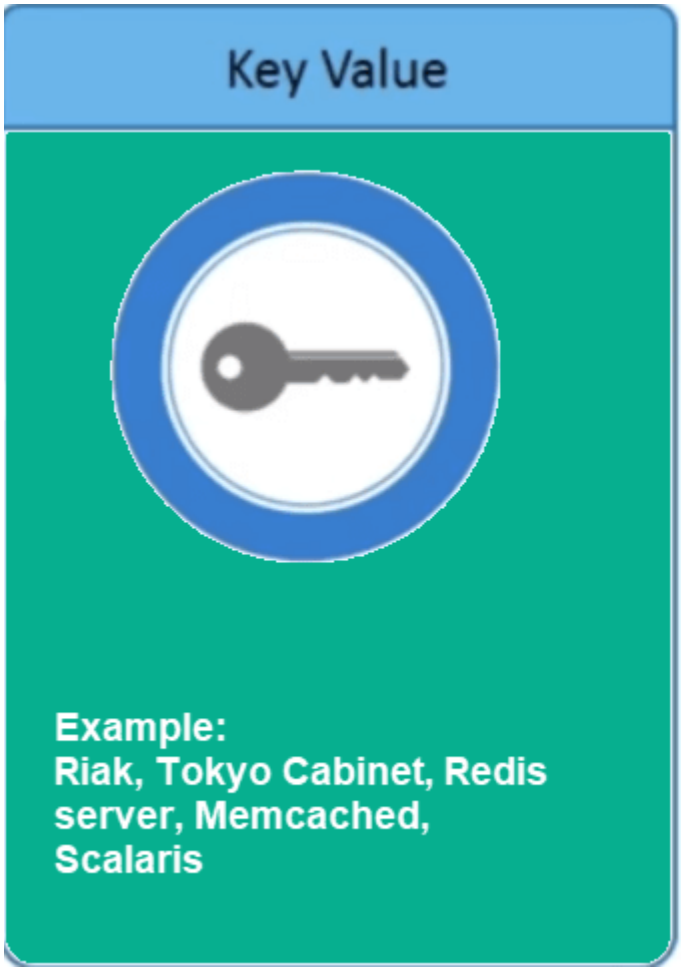


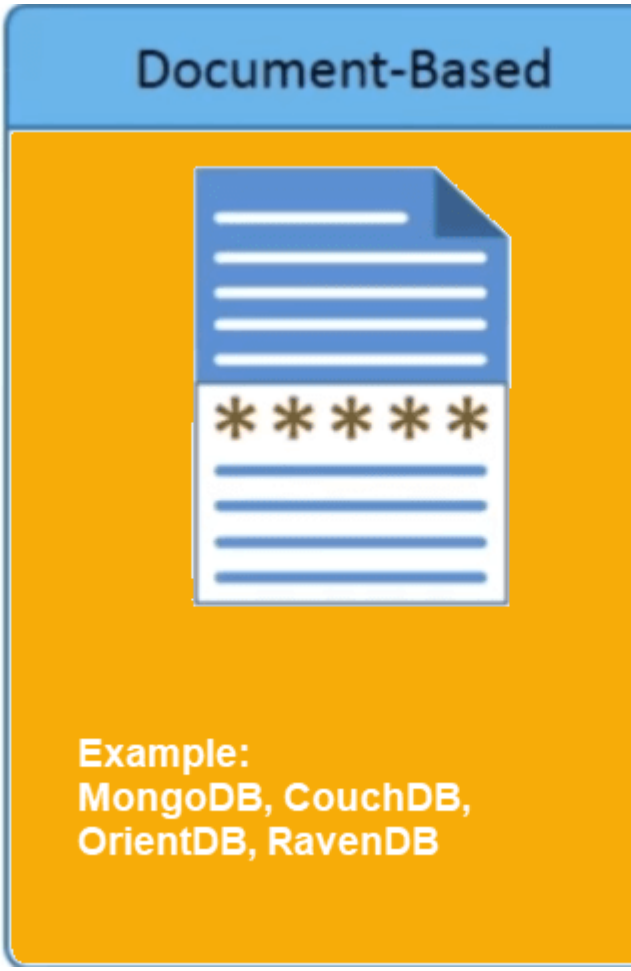
NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.
Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented





Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like “Website” associated with a value like “Guru99”.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based NoSQL database

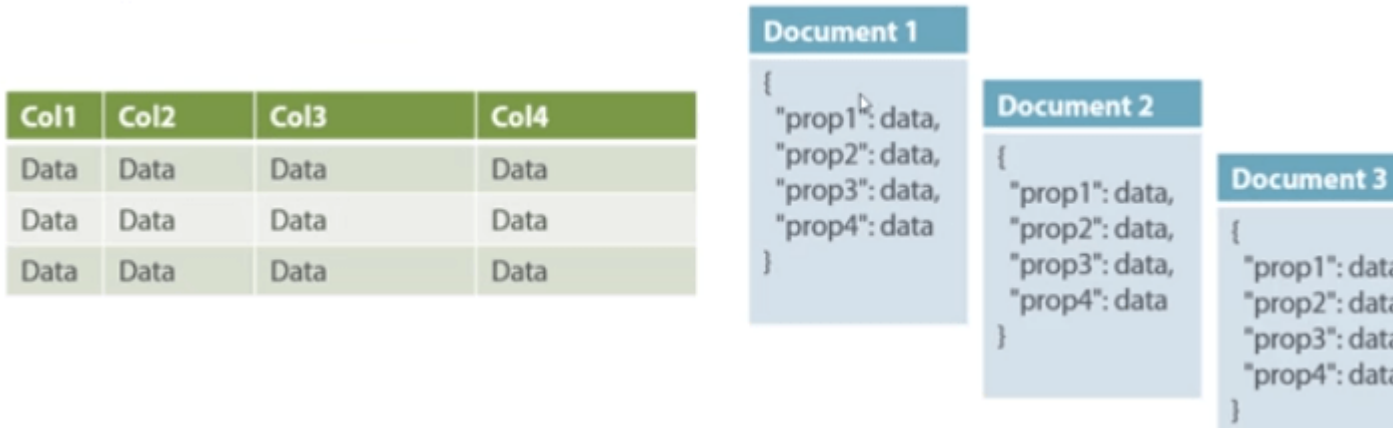
They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

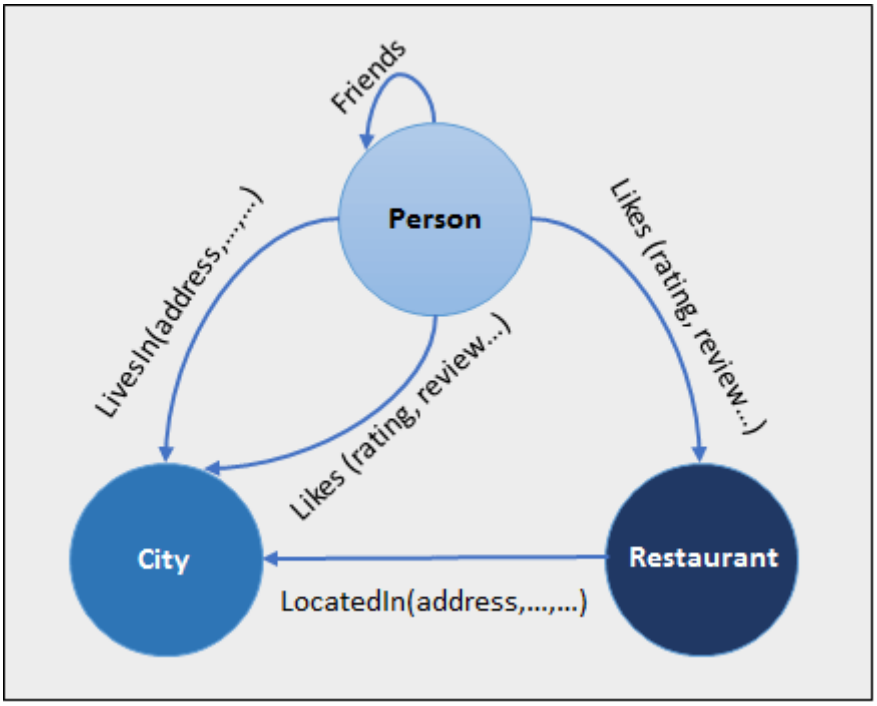
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Query Mechanism tools for NoSQL

The most common data retrieval mechanism is the REST-based retrieval of a value based on its key/ID with GET resource

Document store Database offers more difficult queries as they understand the value in a key-value pair. For example, CouchDB allows defining views with MapReduce

What is the CAP Theorem?

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

1. Consistency
2. Availability
3. Partition Tolerance

Consistency:

The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

Availability:

The database should always be available and responsive. It should not have any downtime.

Partition Tolerance:

Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

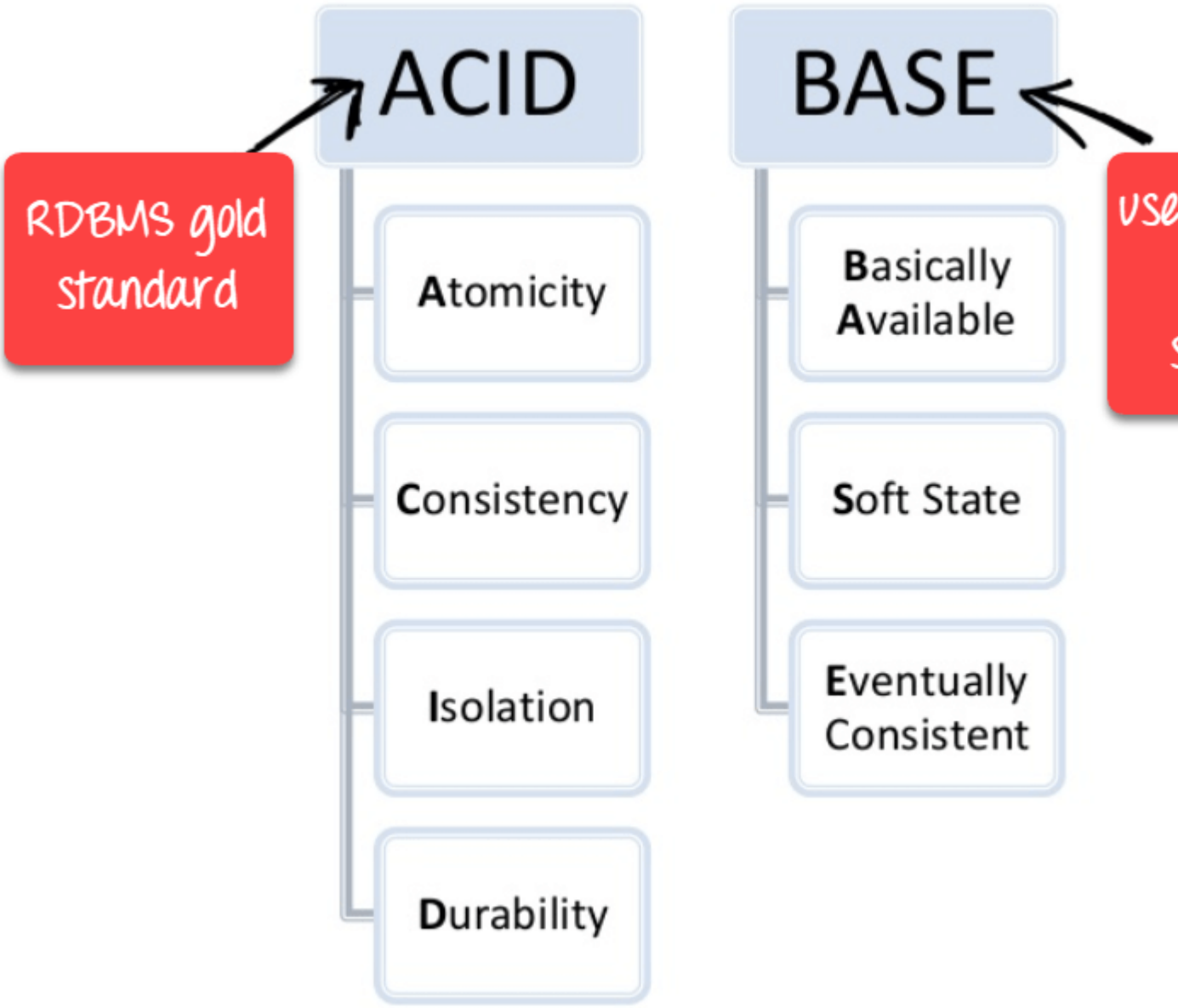
Eventual Consistency

The term “eventual consistency” means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

BASE: **B**asically **A**vailable, **S**oft state, **E**ventual consistency

- Basically, available means DB is available all the time as per CAP theorem
- Soft state means even without an input; the system state may change
- Eventual consistency means that the system will become consistent over time



Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible
- NoSQL databases don’t need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity

- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

Summary

- NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale
- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data
- In the year 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- NoSQL databases never follow the relational model it is either schema-free or has relaxed schemas
- Four types of NoSQL Database are 1). Key-value Pair Based 2). Column-oriented Graph 3). Graphs based 4). Document-oriented
- NOSQL can handle structured, semi-structured, and unstructured data with equal effect
- CAP theorem consists of three words Consistency, Availability, and Partition Tolerance
- BASE stands for **B**asically **A**vailable, **S**oft state, **E**ventual consistency
- The term "eventual consistency" means to have copies of data on multiple machines to get high availability and scalability
- NOSQL offer limited query capabilities