UNIT-I

What is Data

Data is a collection of a distinct small unit of information. It can be used in a variety of forms like tex numbers, media, bytes, etc. it can be stored in pieces of paper or electronic memory, etc.
Word 'Data' is originated from the word 'datum' that means 'single piece of information.' It is plural of th word datum. In computing, Data is information that can be translated into a form for efficient movement ar processing. Data is interchangeable.

What is Database

A database is an organized collection of data, so that it can be easily accessed and managed. You ca organize data into tables, rows, columns, and index it to make it easier to find relevant information. Databas handlers create a database in such a way that only one set of software program provides access of data to a the users. The main purpose of the database is to operate a large amount of information by storing, retrievin and managing data. There are many databases available like MySQL, Sybase, Oracle, MongoDB, Informi PostgreSQL, SQL Server, etc.

Modern databases are managed by the database management system (DBMS).SQL or Structure Query Language is used to operate on the data stored in a database. SQL depends on relational algebra ar tuple relational calculus.A cylindrical structure is used to display the image of a database.

A Database management system is a computerized record-keeping system. It is a repository or container for collection of computerized data files. The overall purpose of DBMS is to allow he users t define, store, retrieve and update the information contained in the database on demand. Information can b anything that is of significance to an individual or organization.

Database System Application

Databases touch all aspects of our lives. Some of the major areas of application are as follows:
Banking, Airlines, Universities, Manufacturing and selling, Human resources, Enterprise Information
Sales: For customer, product, and purchase information.
Accounting: For payments, receipts, account balances, assets and other accounting information.
Human resources: For information about employees, salaries, payroll taxes, and benefits, and for generatio of paychecks.
Manufacturing: For management of the supply chain and for tracking production of items in factorie inventories of items in warehouses and stores, and orders for items.
Online retailers: For sales data noted above plus online order tracking, generation of recommendation list and maintenance of online product evaluations.
Banking and Finance: Banking: For customer information, accounts, loans, and banking transactions.
Credit card transactions: For purchases on credit cards and generation of monthly statements.
Finance: For storing information about holdings, sales, and purchases of financial instruments such as stock and bonds; also for storing real-time market data to enable online trading by customers and automated tradin by the firm.
Universities: For student information, course registrations, and grades (in addition to standard enterpris information such as human resources and accounting).
Airlines: For reservations and schedule information. Airlines were among the first to use databases in geographically distributed manner.
Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances c prepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commerci data. As an example of such methods, typical of the 1960s, consider part of a university organization tha among other data, keeps information about all instructors, students, departments, and course offerings. On way to keep the information on a computer is to store it in operating system files. To allow users t manipulate the information, the system has a number of application programs that manipulate the file including programs to:

Add new students, instructors, and courses Register students for courses and generate class rosters Assign grades to students, compute grade point averages (GPA), and generate transcripts Syste programmers wrote these application programs to meet the needs of the university. New application program are added to the system as the need arises. For example, suppose that a university decides to create a ne major (say, computer science).As a result, the university creates a new department and creates new permaner files (or adds information to existing files) to record information about all the instructors in the departmen students in that major, course offerings, degree requirements, etc. The university may have to write ne application programs to deal with rules specific to the new major. New application programs may also have t be written to handle new rules in the university. Thus, as time goes by, the system acquires more files an more application programs.

This typical file-processing system is supported by a conventional operating system. The system store permanent records in various files, and it needs different application programs to extract records from, an add records to, the appropriate files. Before database management systems (DBMSs) were introduce organizations usually stored information in such systems. Keeping organizational information in a fil processing system has a number of major disadvantages:

Data redundancy and inconsistency. Since different programmers create the files and applicatio programs over a long period, the various files are likely to have different structures and the programs may b written in several programming languages. Moreover, the same information may be duplicated in sever places (files). For example, if a student has a double major (say, music and mathematics) the address an telephone number of that student may appear in a file that consists of student records of students in the Mus department and in a file that consists of student records of students in the Mathematics department. Th redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency; that is, th various copies of the same data may no longer agree. For example, a changed student address may b reflected in the Music department records but not elsewhere in the system. Difficulty in accessing dat

Suppose that one of the university clerks needs to find out the names of all students who live within particular postal-code area. The clerk asks the data-processing department to generate such a list. Because th designers of the original system did not anticipate this request, there is no application program on hand t meet it. There is, however, an application program to generate the list of all students.

The university clerk has now two choices: either obtain the list of all students and extract the neede information manually or ask a programmer to write the necessary application program. Both alternatives ar obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same cler needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of whic is satisfactory. The point here is that conventional file-processing environments do not allow needed data t be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required fo general use.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing ne application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of consistenc constraints. Suppose the university maintains an account for each department, and records the balance amou in each account. Suppose also that the university requires that the account balance of a department may neve fall below zero. Developers enforce these constraints in the system by adding appropriate code in the variou

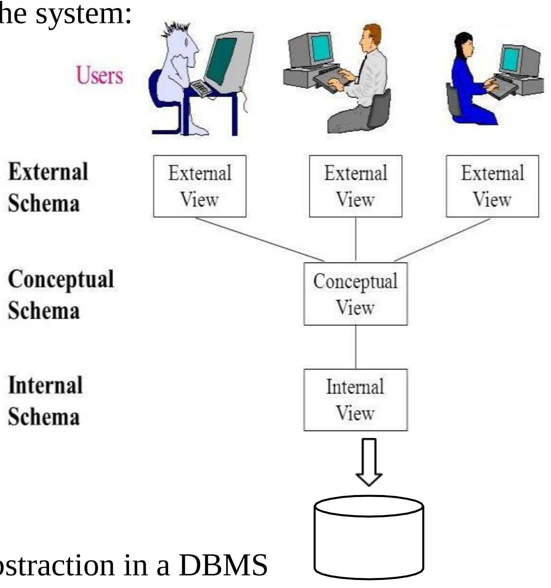
application programs. However, when new constraints are added, it is difficult to change the programs t
enforce them. The problem is compounded when constraints involve several data items from different files.

➢ View of Data

A database system is a collection of interrelated data and a set of programs that allow users to acces
and modify these data. A major purpose of a database system is to provide users with an abstract view of th
data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designe
to use complex data structures to represent data in the database. Since many database-system users are n
computer trained, developers hide the complexity from users through several levels of abstraction, to simplif
users' interactions with the system:



Database DISK

Figure 1.2 : Levels of Abstraction in a DBMS

Physical level (or Internal View / Schema): The lowest level of abstraction describes how the data are actuall
stored. The physical level describes complex low-level data structures in detail.

Logical level (or Conceptual View / Schema): The next-higher level of abstraction describes what data a
stored in the database, and what relationships exist among those data. The logical level thus describes th
entire database in terms of a small number of relatively simple structures. Although implementation of th
simple structures at the logical level may involve complex physical-level structures, the user of the logic
level does not need to be aware of this complexity. This is referred to as physical data independence. Databa
administrators, who must decide what information to keep in the database, use the logical level of abstraction

View level (or External View / Schema): The highest level of abstraction describes only part of the enti
database. Even though the logical level uses simpler structures, complexity remains because of the variety o
information stored in a large database. Many users of the database system do not need all this informatio
instead, they need to access only a part of the database. The view level of abstraction exists to simplify the
interaction with the system. The system may provide many views for the same database. Figure 1.2 shows th
relationship among the three levels of abstraction.

An analogy to the concept of data types in programming languages may clarify the distinction among levels o
abstraction. Many high-level programming languages support the notion of a structured type. For example, w
may describe a record as follows:

type instructor = record
ID : char (5);
name : char (20);

```
dept name : char (20);
salary : numeric (8,2);
end;
```

This code defines a new record type called instructor with four fields. Each field has a name and a typ associated with it. A university organization may have several such record types, including department, with fields dept_name, building, and budget course, with fields course_id, title, dept_name, an credits student, with fields ID, name, dept_name, and tot_cred

At the physical level, an instructor, department, or student record can be described as a block of consecutiv storage locations. The compiler hides this level of detail from programmers. Similarly, the database syster hides many of the lowest-level storage details from database programmers. Database administrators, on th other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous cod segment, and the interrelationship of these record types is defined as well. Programmers using a programmin language work at this level of abstraction. Similarly, database administrators usually work at this level o abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the da types. At the view level, several views of the database are defined, and a database user sees some or all o these views. In addition to hiding details of the logical level of the database, the views also provide a securit mechanism to prevent users from accessing certain parts of the database. For example, clerks in the universit registrar office can see only that part of the database that has information about students; they cannot acces information about salaries of instructors.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of informatio stored in the database at a particular moment is called an instance of the database. The overall design of th database is called the database schema. Schemas are changed infrequently, if at all. The concept of databas schemas and instances can be understood by analogy to a program written in a programming language. database schema corresponds to the variable declarations (along with associated type definitions) in program.

Each variable has a particular value at a given instant. The values of the variables in a program at point in time correspond to an instance of a database schema. Database systems have several schema partitioned according to the levels of abstraction. The physical schema describes the database design at th physical level, while the logical schema describes the database design at the logical level. A database ma also have several schemas at the view level, sometimes called subschemas, which describe different views o the database. Of these, the logical schema is by far the most important, in terms of its effect on applicatio programs, since programmers construct applications by using the logical schema. The physical schema hidden beneath the logical schema, and can usually be changed easily without affecting application program Application programs are said to exhibit physical data independence if they do not depend on the physic schema, and thus need not be rewritten if the physical schema changes.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.
Consider a program to transfer $500 from the account balance of department A to the account balance o department B. If a system failure occurs during the execution of the program, it is possible that the $500 wa removed from the balance of department A but was not credited to the balance of department B, resulting i an inconsistent database state. Clearly, it is essential to database consistency that either both the credit an debit occur, or that neither occur. That is, the funds transfer must be atomic—it must happen in its entirety o not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, mar systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retaile may have millions of accesses per day to their data by shoppers. In such an environment, interaction c concurrent updates is possible and may result in inconsistent data.

Example, suppose a registration program maintains a count of students registered for a course, in ord to enforce limits on the number of students registered. When a student registers, the program reads the curre count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores th count back in the database. Suppose two students register concurrently, with the count at (say) 39. The tw program executions may both read the value 39, and both would then write back 40, leading to an incorre increase of only 1, even though two students successfully registered for the course and the count should be 4 Furthermore, suppose the course registration limit was 40; in the above case both students would be able t register, leading to a violation of the limit of 40 students.

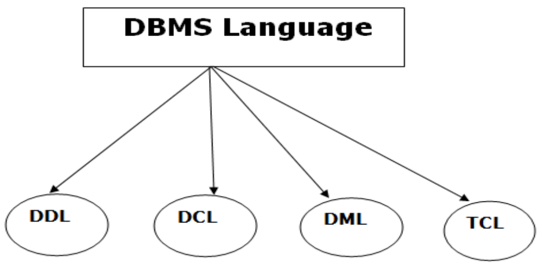
Security problems. Not every user of the database system should be able to access all the data. For exampl in a university, payroll personnel need to see only that part of the database that has financial informatio They do not need access to information about academic records. But, since application programs are added t the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, w shall see the concepts and algorithms that enable database systems to solve the problems with file-processin systems.

➢ Database Languages

Database Language

- 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** stands for **D**ata **D**efinition **L**anguage. 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 ar 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 definitic 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. 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.
  (But in Oracle database, the execution of data control language does not have the feature of rollir back.)

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

➢ Relational Databases:

A relational database is a collection of time-varying, normalized relations of assorted degree
The following intiutitve correspondence can be made

1. A relation is a file
2. Each file contains only one record type
3. The records have no particular order
4. Every field is single-valued
5. The records have a unique identifying field or composite field, called the primary key field

A relational database consists of a collection of tables.
All data values are atomic. No repeating groups are allowed
A relational database is a pointerless database, User does not see or is made aware of pointers.

Relational Database Concepts

Relational database is most commonly used database. It contains number of tables and each table has its ow primary key.
Due to a collection of organized set of tables, data can be accessed easily in RDBMS.
Brief History of RDBMS
During 1970 to 1972, E.F. Codd published a paper to propose the use of relational database model.

RDBMS is originally based on that E.F. Codd's relational model invention.

What is table

The RDBMS database uses tables to store data. A table is a collection of related data entries and contain rows and columns to store data.

A table is the simplest example of data storage in RDBMS.

Let's see the example of student table.

ID	**Name**	**AGE**	**COURSE**
1	Ajeet	24	B.Tech
2	aryan	20	C.A
3	Mahesh	21	BCA
4	Ratan	22	MCA
5	Vimal	26	BSC

What is field

Field is a smaller entity of the table which contains specific information about every record in the table. In th above example, the field in the student table consist of id, name, age, course.

What is row or record

A row of a table is also called record. It contains the specific information of each individual entry in the tabl It is a horizontal entity in the table. For example: The above table contains 5 records.

Let's see one record/row in the table.

s	Ajeet	24	B.Tech

What is column

A column is a vertical entity in the table which contains all information associated with a specific field in table. For example: "name" is a column in the above table which contains all information about student name.

Ajeet
Aryan
Mahesh
Ratan
Vimal

NULL Values

The NULL value of the table specifies that the field has been left blank during record creation. It is totall different from the value filled with zero or a field that contains space.

Data Integrity

There are the following categories of data integrity exist with each RDBMS:

Entity integrity: It specifies that there should be no duplicate rows in a table.

Domain integrity: It enforces valid entries for a given column by restricting the type, the format, or the rang of values.

Referential integrity: It specifies that rows cannot be deleted, which are used by other records.

User-defined integrity: It enforces some specific business rules that are defined by users. These rules a different from entity, domain or referential integrity

➢ Database Design

Database design can be generally defined as a collection of tasks or processes that enhance th designing, development, implementation, and maintenance of enterprise data management system. Designin a proper database reduces the maintenance cost thereby improving data consistency and the cost-effectiv measures are greatly influenced in terms of disk storage space. Therefore, there has to be a brilliant concept o designing a database. The designer should follow the constraints and decide how the elements correlate an what kind of data must be stored.

The main objectives behind database designing are to produce physical and logical design models c the proposed database system. To elaborate this, the logical model is primarily concentrated on th requirements of data and the considerations must be made in terms of monolithic considerations and hence th stored physical data must be stored independent of the physical conditions. On the other hand, the physic database design model includes a translation of the logical design model of the database by keep control c physical media using hardware resources and software systems such as Database Management Syster (DBMS).

Why is Database Design important?

The important consideration that can be taken into account while emphasizing the importance of databas design can be explained in terms of the following points given below.

1. Database designs provide the blueprints of how the data is going to be stored in a system. A prope design of a database highly affects the overall performance of any application.
2. The designing principles defined for a database give a clear idea of the behavior of any application an how the requests are processed.
3. Another instance to emphasize the database design is that a proper database design meets all th requirements of users.
4. Lastly, the processing time of an application is greatly reduced if the constraints of designing a highl efficient database are properly implemented.

Requirement Analysis

First of all, the planning has to be done on what are the basic requirements of the project under which th design of the database has to be taken forward. Thus, they can be defined as:-

Planning - This stage is concerned with planning the entire DDLC (Database Development Life Cycle). Th strategic considerations are taken into account before proceeding.

System definition - This stage covers the boundaries and scopes of the proper database after planning.

Database Designing

The next step involves designing the database considering the user-based requirements and splitting them ou into various models so that load or heavy dependencies on a single aspect are not imposed. Therefore, ther has been some model-centric approach and that's where logical and physical models play a crucial role.

Physical Model - The physical model is concerned with the practices and implementations of the logic model.

Logical Model - This stage is primarily concerned with developing a model based on the propose requirements. The entire model is designed on paper without any implementation or adopting DBM considerations.

Implementation

The last step covers the implementation methods and checking out the behavior that matches ou requirements. It is ensured with continuous integration testing of the database with different data sets an conversion of data into machine understandable language. The manipulation of data is primarily focused c these steps where queries are made to run and check if the application is designed satisfactorily or not.

Data conversion and loading - This section is used to import and convert data from the old to the ne system.

Testing - This stage is concerned with error identification in the newly implemented system. Testing is crucial step because it checks the database directly and compares the requirement specifications.

➢ Database Design Process

The process of designing a database carries various conceptual approaches that are needed to be kept in min
An ideal and well-structured database design must be able to:

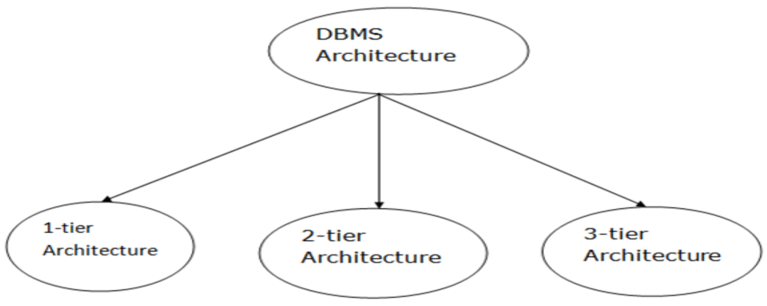
1. Save disk space by eliminating redundant data.
2. Maintains data integrity and accuracy.
3. Provides data access in useful ways.
4. Comparing Logical and Physical data models.

➢ Database Architecture

- The DBMS design depends upon its architecture. The basic client/server architecture is used to de with a large number of PCs, web servers, database servers and other components that are connect with networks.
- The client/server architecture consists of many PCs and a workstation which are connected via th network.
- DBMS architecture depends upon how users are connected to the database to get their request done.

Types of DBMS Architecture

Database architecture can be seen as a single tier or multi-tier. But logically, database architecture is of two types like: **2-tier architecture** and **3-tier architecture**.



1-Tier Architecture

- In this architecture, the database is directly available to the user. It means the user can directly sit on the DBMS and uses it.
- Any changes done here will directly be done on the database itself. It doesn't provide a handy tool f end users.
- The 1-Tier architecture is used for development of the local application, where programmers c directly communicate with the database for the quick response.

2-Tier Architecture

- The 2-Tier architecture is same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. For this interaction, API's like: **ODBC**, **JDBC** are used.
- The user interfaces and application programs are run on the client-side.
- The server side is responsible to provide the functionalities like: query processing and transaction management.
- To communicate with the DBMS, client-side application establishes a connection with the server side.

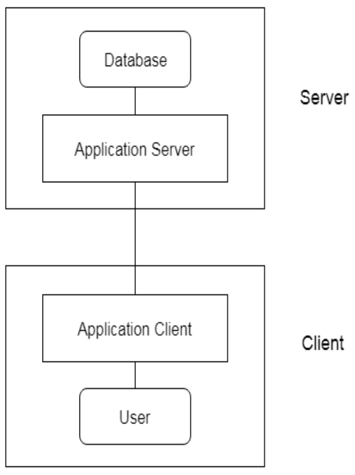
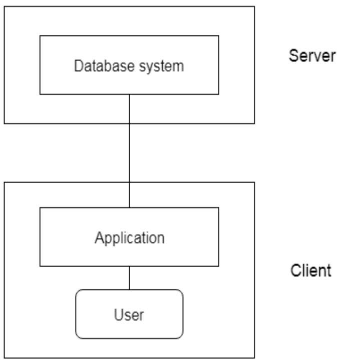


Fig: 2-tier Architecture

3-Tier Architecture

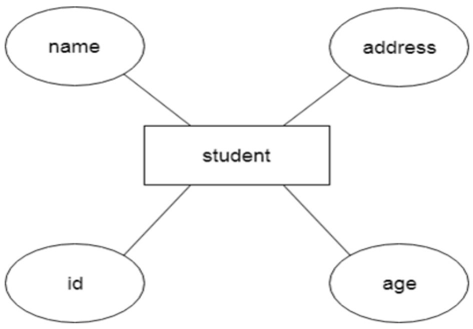


- The 3-Tier architecture contains another layer between the clie and server. In this architecture, client can't directly communicate wi the server.
- The application on the client-end interacts with an applicatio server which further communicates with the database system.
- End user has no idea about the existence of the database beyon the application server. The database also has no idea about any othe user beyond the application.
- The 3-Tier architecture is used in case of large web application.

Fig: 3-tier Architecture

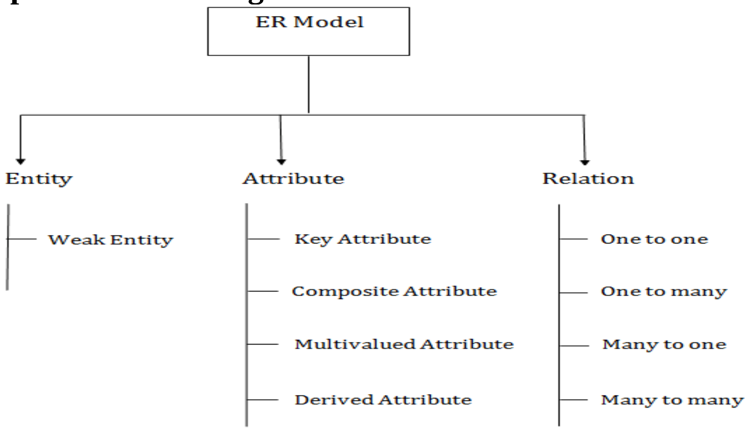
➢ ER model(Entity Relationship)

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.



For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin cod etc and there will be a relationship between them.

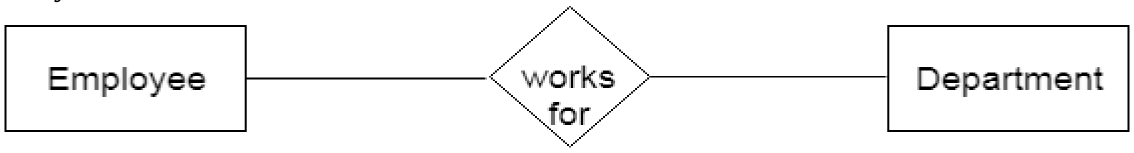
Component of ER Diagram



1. Entity:

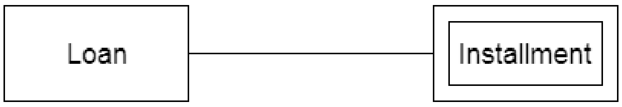
An entity may be any object, class, person or place. In the ER diagram, an entity can be represented a rectangles.

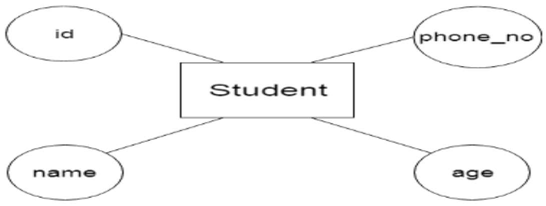
Consider an organization as an example- manager, product, employee, department etc. can be taken as a entity.



a. Weak Entity

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any ke attribute of its own. The weak entity is represented by a double rectangle.





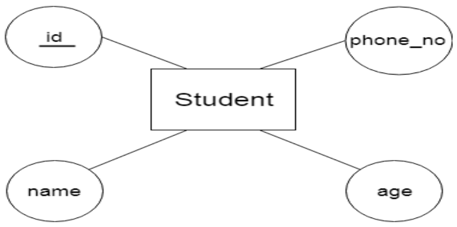
2. Attribute

The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.

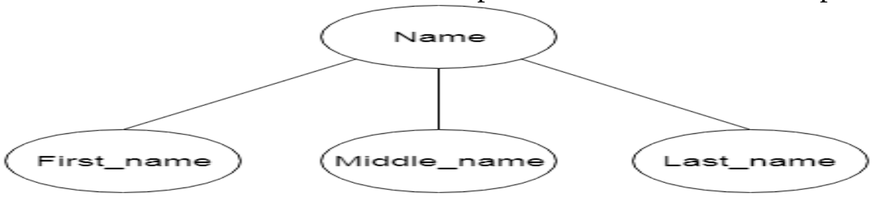
a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



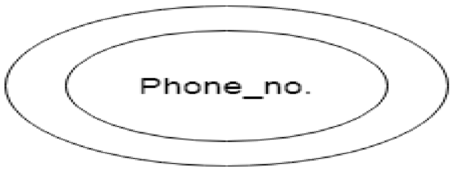
b. Composite Attribute

An attribute that composed of many other attributes is known as a composite attribute. The composi attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. Multivalued Attribute

An attribute can have more than one value. These attributes are know as a multivalued attribute. The double oval is used to represent multivalued attribute.

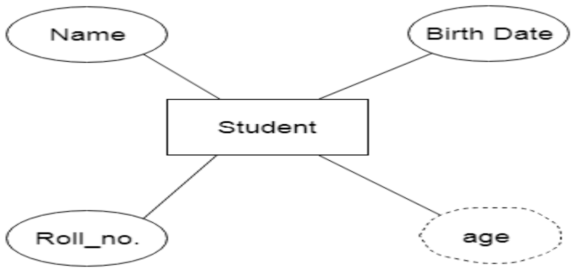


For example, a student can have more than one phone number.

d. Derived Attribute

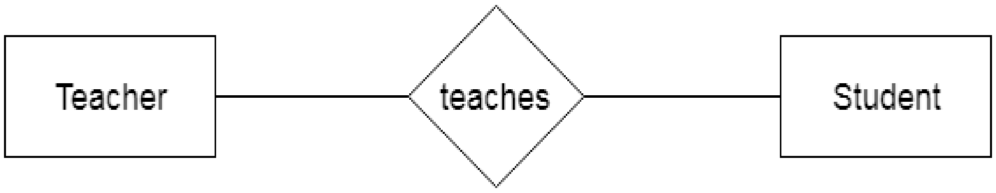
An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.



3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent th relationship.

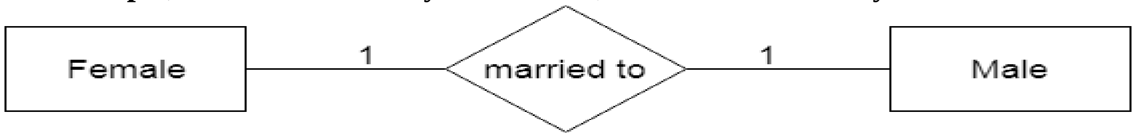


Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to or relationship.

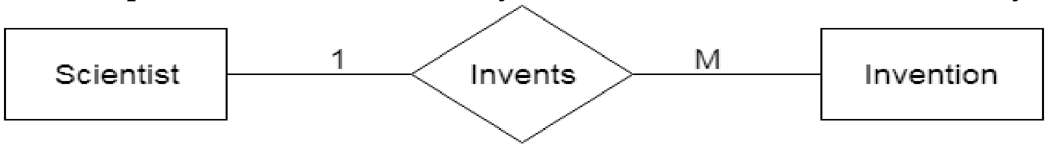
For example, A female can marry to one male, and a male can marry to one female.



b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the rigl associates with the relationship then this is known as a one-to-many relationship.

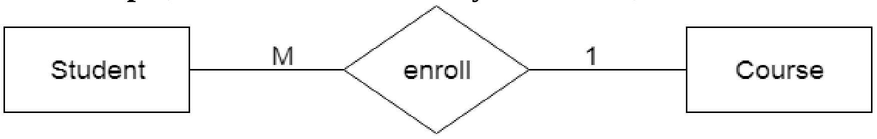
For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.



c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the rigl associates with the relationship then it is known as a many-to-one relationship.

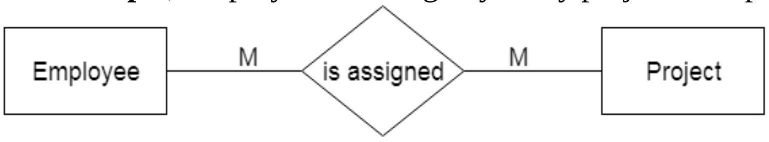
For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the rigl associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.



Notation of ER diagram

Database can be represented using the notations. In ER diagram, many notations are used to express th cardinality. These notations are as follows:

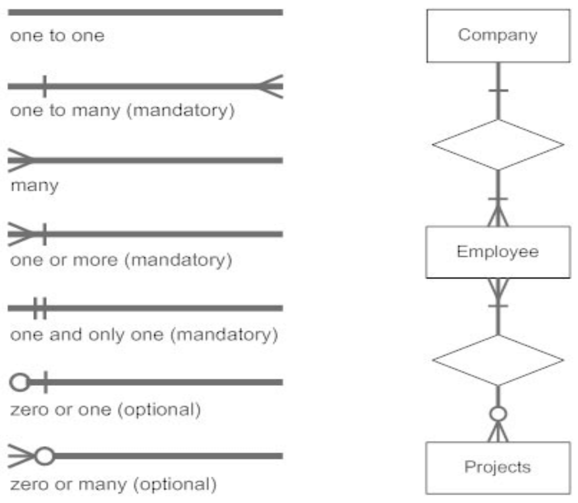
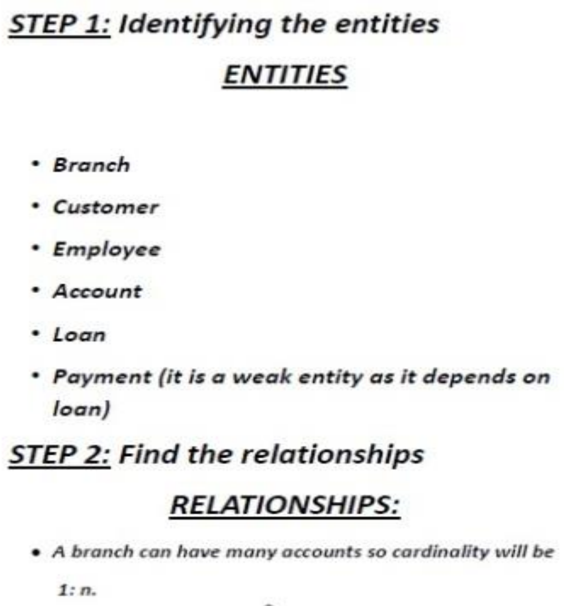


Fig: Notations of ER diagram

Database Design for Banking Enterprise and Unified Modeling Language. E-R Diagram for banking enterprise

A bank has many branches in a country. Each branch is identified with the help of branch name. Each branch of the bank has multiple customers. Customers also include employees of that particular branch. An employee can be a manager or staff of that branch. Branch gives each customer an id in order to identify them. Customers can open two types of accounts i.e. saving account and Checking account. Branches of the bank provide Loan to the customers if required. Re-Payment of loan is done with the help of payment number of that particular loan. Draw an extended ER diagram with the help of above details.



<u>STEP 1:</u> Identifying the entities

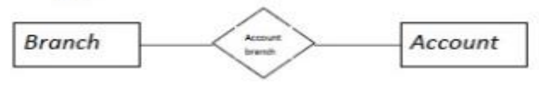
<u>ENTITIES</u>

- ***Branch***
- ***Customer***
- ***Employee***
- ***Account***
- ***Loan***
- ***Payment (it is a weak entity as it depends on loan)***

<u>STEP 2:</u> Find the relationships

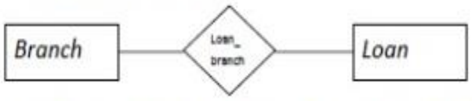
<u>RELATIONSHIPS:</u>

- ***A branch can have many accounts so cardinality will be 1: n.***

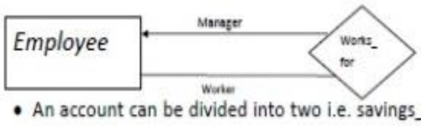


- Branches of bank give loans to the customers.

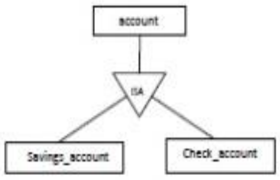
  Cardinality will be 1: n



- Employee can be a manager or a worker of that particular branch

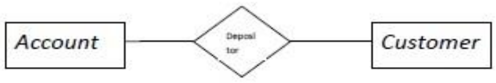


- An account can be divided into two i.e. savings_account and check_account.



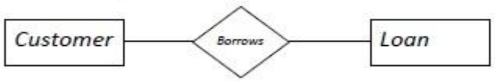
- Customers will be having account in different branches

  Cardinality will be n: 1.



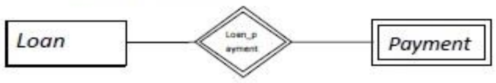
- Customers are allowed to take loan from the bank.

  Cardinality will be 1: n.



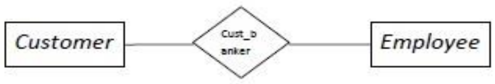
- Loan is paid back with the help of payment number.

  Cardinality will be 1: n.



- A customer can be an employee of the bank.

  Cardinality will be n: m.



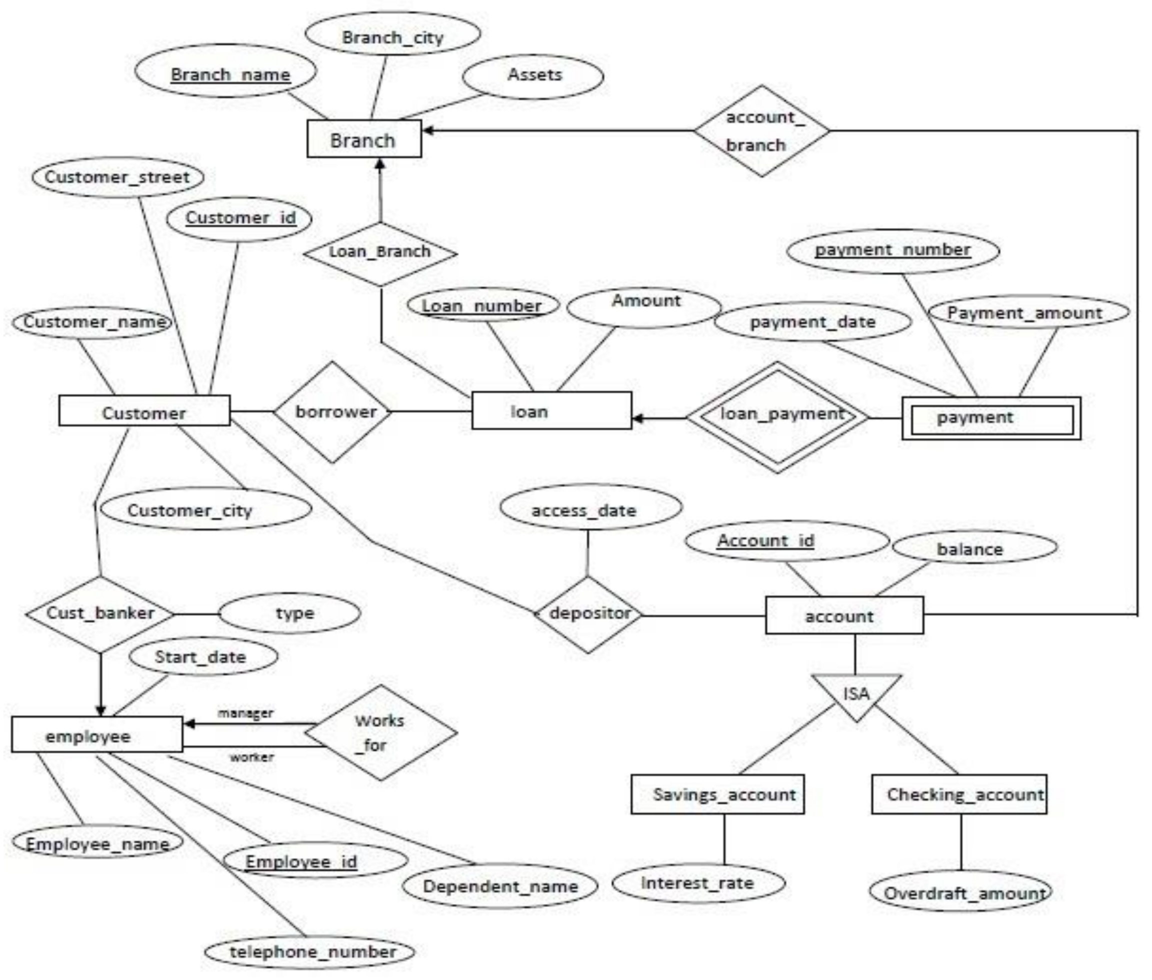
STEP3: ***Identify the primary key***

Entity	Primary key
Branch	Branch_name
Customer	Customer_id
Loan	Loan_number
Employee	Employee_id
Account	Account_id
Payment	Payment_number

Step4: ***Identify another attributes***

Entity	*Other*	*entities*			
Branch	*Branch_name*	*Branch_city*	*Asset*		
Account	*Account_name*	*Balance*			
Customer	*Customer_id*	*Customer_name*	*Customer_street*	*Customer_city*	
Loan	*Loan_number*	*Amount*			
Payment	*Payment_number*	*Payment_date*	*Payment_amount*		
Employee	*Employee_id*	*Employee_name*	*Telephone_number*	*Dependent_name*	*Start_date*

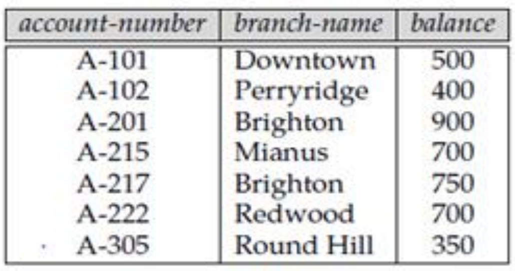
Step 5: Complete the ER diagram.



➢ Structure of Relational Database

A relational database consists of a collection of **tables**, each of which is assigned a unique name.

A row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name. In what follows, we introduce the concept of relation.



account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure 3.1 The *account* relation.

Basic Structure

Consider the account table of Figure 3.1. It has three column headers: account-number, branch-name, and balance. Following the terminology of the relational model, these headers are **attributes**. For each attribute, there is a set of permitted values, called

the **domain** of that attribute. For the attribute branch-name, for example, the domain is the set of all branch names.
LetD1 denote the set of all account numbers, D2 the set of all branch names, and D3the set of all balances. Any row of account must consist of a 3-tuple (v1, v2, v3), where v1 is an account number (that is, v1 is in domain D1),v2 is a branch name (that is, v2 is in domain D2), and v3 is a balance (that is, v3 is in domain D3). In general, account will contain only a subset of the set of all possible rows. Therefore,

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Figure 3.2 The *account* relation with unordered tuples.

account is a subset of

$$D_1 \times D_2 \times D_3$$

In general, a **table** of n attributes must be a subset of

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

Mathematicians define a **relation** to be a subset of a Cartesian product of a list of domains. This definition corresponds almost exactly with our definition of table. The only difference is that we have assigned names to attributes, whereas mathematicians rely on numeric "names," using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the attribute whose domain appears second, and so on. Because tables are essentially relations, we shall use the mathematical terms relation and tuple in place of the terms table and row. A tuple variable is a variable that stands for a tuple; in other words, a tuple variable is a variable whose domain is the set of all tuples.

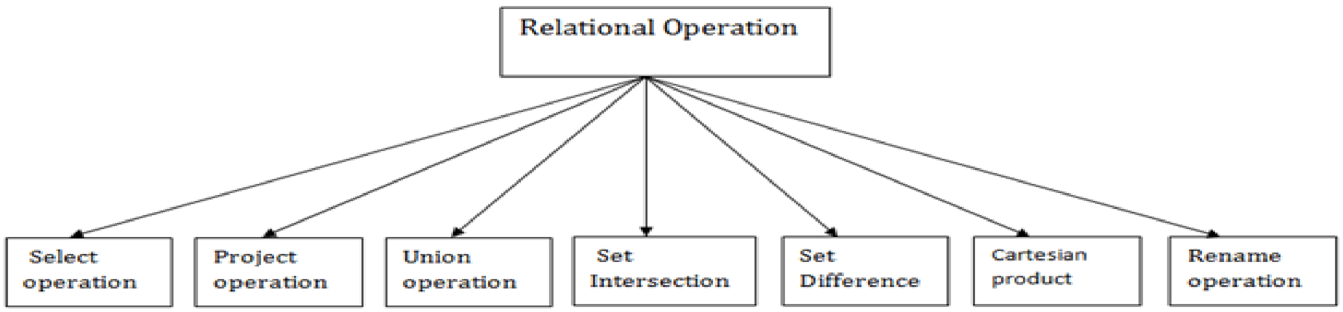
In the account relation of Figure 3.1, there are seven tuples. Let the tuple variable t refer to the first tuple of the relation. We use the notation t[account-number] to denote the value of t on the account-number attribute. Thus, t[account-number] = "A-101," and t[branch-name] = "Downtown". Alternatively, we may write t[1] to denote the value of tuple t on the first attribute (account-number), t[2] to denote branch-name, and so on. Since a relation is a set of tuples, we use the mathematical notation of t ∈r to denote that tuple t is in relation r.

The order in which tuples appear in a relation is irrelevant, since a relation is a set of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 3.1, or are unsorted, as in Figure 3.2, does not matter; the relations in the two figures above are the same, since both contain the same set of tuples.

For all relations r, the domains of all attributes of r be atomic. A domain is atomic if elements of the domain are considered to be indivisible units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a non atomic domain.

➢ Relational Algebra

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



1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma (σ).

1. Notation: σ p(r)

Where:σ is used for selection prediction
r is used for relation
p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

For example: LOAN Relation

BRANCH_NAME	**LOAN_NO**	**AMOUNT**
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

1. σ BRANCH_NAME="perryride" (LOAN)

Output:

BRANCH_NAME	**LOAN_NO**	**AMOUNT**
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

- This operation shows the list of those attributes that we wish to appear in the result. Rests of the attributes are eliminated from the table.
- It is denoted by ∏.

1. Notation: ∏ A1, A2, An (r)

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

Example: CUSTOMER RELATION

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

Input:

1. ∏ NAME, CITY (CUSTOMER)

Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

3. Union Operation:

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

1. **Notation: R ∪ S**

A union operation must hold the following condition:

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

Example:

DEPOSITOR RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Input:

1. ∏ CUSTOMER_NAME (BORROW) ∪ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Mayes

4. Set Intersection:

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

1. Notation: R ∩ S

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. ∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Smith
Jones

5. Set Difference:

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

1. Notation: R - S

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. ∏ CUSTOMER_NAME (BORROW) - ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Jackson
Hayes
Willians
Curry

6. Cartesian product

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

1. Notation: E X D

Example:

EMPLOYEE

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B

DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

Input:

1. EMPLOYEE X DEPARTMENT

Output:

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

7. Rename Operation:

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

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

Modification of the Database

The SQL Modification Statements make changes to database data in tables and columns. There are 3 modification statements:

- INSERT Statement -- add rows to tables
- UPDATE Statement -- modify columns in table rows
- DELETE Statement -- remove rows from tables

INSERT Statement

The INSERT Statement adds one or more rows to a table. It has two formats:

INSERT INTO table-1 [(column-list)] VALUES (value-list)

and,

INSERT INTO table-1 [(column-list)] (query-specification)

VALUES Clause

The VALUES Clause in the INSERT Statement provides a set of values to place in the columns of a new row. It has the following general format:

VALUES (value-1 [, value-2] ...)

value-1 and value-2 are Literal Values or Scalar Expressions involving literals. They can also specify NULL.

The values list in the VALUES clause must match the explicit or implicit column list for INSERT in degree (number of items). They must also match the data type of corresponding column or be convertible to that data type.

INSERT Examples

INSERT INTO p (pno, color) VALUES ('P4', 'Brown')

Before

pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green

=>

After

pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green
P4	NULL	Brown

INSERT INTO sp
SELECT s.sno, p.pno, 500
FROM s, p
WHERE p.color='Green' AND s.city='London'

Before

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

After

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200
S2	P3	500

UPDATE Statement

The UPDATE statement modifies columns in selected table rows. It has the following general format:

UPDATE Examples

UPDATE sp SET qty = qty + 20

Before **After**

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

sno	pno	qty
S1	P1	NULL
S2	P1	220
S3	P1	1020
S3	P2	220

UPDATE s
SET name = 'Tony', city = 'Milan'
WHERE sno = 'S3'

Before

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

=>

After

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Tony	Milan

DELETE Statement

The DELETE Statement removes selected rows from a table. It has the following general format:

DELETE FROM table-1 [WHERE predicate]

The optional WHERE Clause has the same format as in the SELECT Statement. See WHERE Clause. The WHERE clause chooses which table rows to delete. If it is missing, all rows are in table-1 are removed. The WHERE Clause predicate can contain subqueries, but the subqueries cannot reference table-1. This prevents situations where results are dependent on the order of processing.

DELETE Examples

DELETE FROM sp WHERE pno = 'P1'

Before

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

After

sno	pno	qty
S3	P2	200

DELETE FROM p WHERE pno NOT IN (SELECT pno FROM sp)

Before

pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green

=>

After

pno	descr	color
P1	Widget	Blue
P2	Widget	Red

UNIT-II

- **SQL**: SQL is a short-form of the structured query language, and it is pronounced as S-Q-L o sometimes as See-Quell.

This database language is mainly designed for maintaining the data in relational database manageme systems. It is a special tool used by data professionals for handling structured data (data which is stored in th form of tables). It is also designed for stream processing in RDSMS.
You can easily create and manipulate the database, access and modify the table rows and columns, etc. Th query language became the standard of ANSI in the year of 1986 and ISO in the year of 1987.
If you want to get a job in the field of data science, then it is the most important query language to learn. Bi enterprises like Facebook, Instagram, and LinkedIn, use SQL for storing the data in the back-end.

- **Structure of SQL Queries:**

SQL queries are the questions or requests imposed on the set of data to retrieve the desire information. The language we use to build these queries is structured query language i.e. SQL. In this contex we will be discussing the structure of SQL queries and try to understand them with some examples.
What is the Basic Structure of SQL Queries?
The fundamental structure of SQL queries includes three clauses that are select, from, and where clause. Wha we want in the final result relation is specified in the select clause. Which relations we need to access to g the result is specified in from clause. How the relation must be operated to get the result is specified i the where clause. select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;
In the select clause, you have to specify the attributes that you want to see in the result relation
In the from clause, you have to specify the list of relations that has to be accessed for evaluating the query.
In the where clause involves a predicate that includes attributes of the relations that we have listed in the from clause.

Though the SQL query has a sequence select, from, and where. To understand how the query will operate? You must consider the query in the order, from, where and then focus on select.
So with the help of these three clauses, we can retrieve the information we want out of the huge set of data

SQL Syntax

When you want to do some operations on the data in the database, then you must have to write the query in the predefined syntax of SQL. The syntax of the structured query language is a unique set of rules and guidelines, which is not case-sensitive. Its Syntax is defined and maintained by the ISO and ANSI standards. Following are some most important points about the SQL syntax which are to remember: You can write the keywords of SQL in both uppercase and lowercase, but writing the SQL keywords in uppercase improves the readability of the SQL query.
Simple Example of SQL statement:
SELECT "column name" FROM "table name";
Each SQL statement begins with any of the SQL keywords and ends with the semicolon (;). The semicolon is used in the SQL for separating the multiple Sql statements which are going to execute in the same call.
Let's discuss each statement in short one by one with syntax and one example:
1. SELECT Statement
This SQL statement reads the data from the SQL database and shows it as the output to the database user.
Syntax of SELECT Statement:

SELECT column_name1, column_name2, …., column_nameN
[FROM table name] [WHERE condition]
[ORDER BY order_column_name1 [ASC | DESC], ...];

Example of SELECT Statement:

SELECT Emp_ID, First Name, Last Name, Salary, City
FROM Employee_details
WHERE Salary = 100000
ORDER BY Last_Name

This example shows the Emp_ID, First Name, Last_Name, Salary, and City of those employees from the Employee_details table whose Salary is 100000. The output shows all the specified details according to the ascending alphabetical order of Last_Name.

UPDATE Statement

This SQL statement changes or modifies the stored data in the SQL database.

Syntax of UPDATE Statement:

UPDATE table_name
SET column_name1= new_value_1, column_name2 = new_value_2, ..., column_nameN= new_value_N
[WHERE CONDITION];

Example of UPDATE Statement:

UPDATE Employee_details
SET Salary = 100000
WHERE Emp_ID = 10;

This example changes the Salary of those employees of the Employee_details table whose Emp_ID is 10 in the table.

DELETE Statement

This SQL statement deletes the stored data from the SQL database.

Syntax of DELETE Statement:

DELETE FROM table_name [WHERE CONDITION];

Example of DELETE Statement:DELETE FROM Employee_details
WHERE First_Name = 'Sumit';

This example deletes the record of those employees from the Employee_details table whose First_Name is Sumit in the table.

CREATE TABLE Statement

This SQL statement creates the new table in the SQL database.

Example of CREATE TABLE Statement:

CREATE TABLE Employee_details (Emp_Id NUMBER(4) NOT NULL, First_name VARCHAR (30),Last_name VARCHAR(30),Salary Money, City VARCHAR(30) PRIMARY KEY (Emp_Id));

The fields in the table are Emp_Id, First_Name, Last_Name, Salary, and City. The Emp_Id column in the table acts as a primary key, which means that the Emp_Id column cannot contain duplicate values and null values.

ALTER TABLE Statement

This SQL statement adds, deletes, and modifies the columns of the table in the SQL database.

Syntax of ALTER TABLE Statement:

ALTER TABLE table_name ADD column_name datatype[(size)];

The above SQL alter statement adds the column with its datatype in the existing database table.

ALTER TABLE table_name MODIFY column_name column_datatype[(size)];

The above 'SQL alter statement' renames the old column name to the new column name of the existing database table.
ALTER TABLE table_name DROP COLUMN column_name;
The above SQL alter statement deletes the column of the existing database table.
Example of ALTER TABLE Statement:
ALTER TABLE Employee_details ADD Designation VARCHAR (18);
This example adds the new field whose name is Designation with size 18 in the Employee_details table of the SQL database.

DROP TABLE Statement

This SQL statement deletes or removes the table and the structure, views, permissions, and triggers associated with that table.
Syntax of DROP TABLE Statement: DROP TABLE [IF EXISTS]
table_name1, table_name2, ……, table_nameN;
The above syntax of the drop statement deletes specified tables completely if they exist in the database.
Example of DROP TABLE Statement:
DROP TABLE Employee_details;
This example drops the Employee_details table if it exists in the SQL database. This removes the complete information if available in the table.

INSERT INTO Statement

This SQL statement inserts the data or records in the existing table of the SQL database. This statement can easily insert single and multiple records in a single query statement.
Syntax of insert a single record:
INSERT INTO table_name (column_name1, column_name2, …., column_nameN)
VALUES (value_1, value_2, ..…, value_N);
Example of insert a single record:
INSERT INTO Employee_details (Emp_ID, First_name, Last_name, Salary, City)
VALUES (101, Akhil, Sharma, 40000, Bangalore);
This example inserts 101 in the first column, Akhil in the second column, Sharma in the third column, 40000 in the fourth column, and Bangalore in the last column of the table Employee_details.
Syntax of inserting a multiple records in a single query:
INSERT INTO table_name (column_name1, column_name2, …., column_nameN)
VALUES (value_1, value_2, ..…, value_N), (value_1, value_2, ..…, value_N),….;
Example of inserting multiple records in a single query:
INSERT INTO Employee_details (Emp_ID, First_name, Last_name, Salary, City)
VALUES (101, Amit, Gupta, 50000, Mumbai), (101, John, Aggarwal, 45000, Calcutta), (101, Sidhu, Aror a, 55000, Mumbai);
This example inserts the records of three employees in the Employee_details table in the single query statement.

TRUNCATE TABLE Statement

This SQL statement deletes all the stored records from the table of the SQL database.
Syntax of TRUNCATE TABLE Statement:
TRUNCATE TABLE table_name;
Example of TRUNCATE TABLE Statement:
TRUNCATE TABLE Employee_details;
This example deletes the record of all employees from the Employee_details table of the database.

DESCRIBE Statement

This SQL statement tells something about the specified table or view in the query.
Syntax of DESCRIBE Statement:
DESCRIBE table_name | view name;
Example of DESCRIBE Statement:
DESCRIBE Employee_details;
This example explains the structure and other details about the Employee_details table.

DISTINCT Clause

This SQL statement shows the distinct values from the specified columns of the database table. This statement is used with the SELECT keyword.
Syntax of DISTINCT Clause:
SELECT DISTINCT column_name1, column_name2, ...
FROM table_name;
Example of DISTINCT Clause:
SELECT DISTINCT City, Salary
FROM Employee_details;
This example shows the distinct values of the City and Salary column from the Employee_details table.

COMMIT Statement

This SQL statement saves the changes permanently, which are done in the transaction of the SQL database.
Syntax of COMMIT Statement:COMMIT
Example of COMMIT Statement:
DELETE FROM Employee_details
WHERE salary = 30000;
COMMIT;
This example deletes the records of those employees whose Salary is 30000 and then saves the changes permanently in the database.

ROLLBACK Statement

This SQL statement undoes the transactions and operations which are not yet saved to the SQL database.
Syntax of ROLLBACK Statement:
ROLLBACK
Example of ROLLBACK Statement:
DELETE FROM Employee_details
WHERE City = Mumbai;
ROLLBACK;
This example deletes the records of those employees whose City is Mumbai and then undo the changes in the database.

CREATE INDEX Statement

This SQL statement creates the new index in the SQL database table.
Syntax of CREATE INDEX Statement:
CREATE INDEX index name
ON table_name (column_name1, column_name2, …, column_nameN);
Example of CREATE INDEX Statement:
CREATE INDEX idx_First_Name
ON employee details (First Name);
This example creates an index idx_First_Name on the First Name column of the Employee_details table.

DROP INDEX Statement

This SQL statement deletes the existing index of the SQL database table.

Syntax of DROP INDEX Statement:
DROP INDEX index name;
Example of DROP INDEX Statement:
DROP INDEX idx_First_Name;
This example deletes the index idx_First_Name from the SQL database

➢ SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.
Types of Set Operation
Union
UnionAll
Intersect
Minus

Union
Union All
1
2
Set Operation
3
4
Intersect
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
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
Example:
The First tableOPs Concepts in Java

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:
SELECT * FROM First
UNION
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:

SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
Example: Using the above First and Second table.
Union All query will be like:
SELECT * FROM First
UNION ALL
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
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
Example:
Using the above First and Second table.
Intersect query will be:
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
The resultset table will look like:

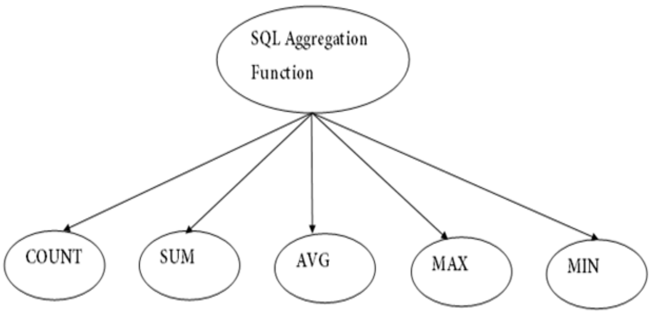
ID	NAME
3	Jackson

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:
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
Example
Using the above First and Second table.
Minus query will be:
SELECT * FROM First
MINUS

SELECT * FROM Second;
The resultset table will look like:

ID	NAME
1	Jack
2	Harry

➢ 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
COUNT(*) or
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	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: COUNT() OPs Concepts in Java
SELECT COUNT(*)
FROM PRODUCT_MAST;
Output: 10
Example: COUNT with WHERE
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE>=20;
Output:7
Example: COUNT() with DISTINCT
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
Output:3
Example: COUNT () with GROUP BY
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST

```
GROUP BY COMPANY;
```
Output:
```
Com1    5
Com2    3
Com3    2
```
Example: COUNT() with HAVING
```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY
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
```
SUM()  or
SUM( [ALL|DISTINCT] expression )
```
Example: SUM()
```
SELECT SUM (COST)  FROM PRODUCT_MAST;
```
Output: 670

Example: SUM() with WHERE SELECT SUM(COST)
FROM PRODUCT_MAST WHERE QTY>3;

Output:320

Example: SUM() with GROUP BY
```
SELECT SUM(COST)  FROM PRODUCT_MAST  WHERE QTY>3
GROUP BY COMPANY;
```
Output:
```
Com1    150
Com2    170
```
Example: SUM() with HAVING
```
SELECT COMPANY, SUM(COST)  FROM PRODUCT_MAST
GROUP BY COMPANY  HAVING SUM(COST)>=170;
```
Output:
```
Com1    335
Com3    170
```
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
```
AVG()  or  AVG( [ALL|DISTINCT] expression )
```
Example:
```
SELECT AVG(COST)  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

MAX() or MAX([ALL|DISTINCT] expression)

Example:

SELECT MAX(RATE) FROM PRODUCT_MAST; **Output: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

MIN () or MIN ([ALL|DISTINCT] expression)

Example:

SELECT MIN(RATE) FROM PRODUCT_MAST;

Output: 10

➢ Nested Sub Queries

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**. Query 4 is for- mulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator IN, which compares a value *v* with a set (or multiset) of values *V* and evaluates to TRUE if *v* is one of the elements in *V*.

The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

```
SELECT      DISTINCT Pnumber
FROMPROJECT
WHERE       Pnumber IN
(SELECT     Pnumber
FROM        PROJECT, DEPARTMENT, EMPLOYEE
WHERE       Dnum=Dnumber AND
Mgr_ssn=Ssn AND Lname='Smith')
                                OR
                                Pnumber IN
                                (SELECT     Pno
                                 FROM       WORKS_ON, EMPLOYEE

                                 WHERE      Essn=Ssn AND Lname='Smith');
```

If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a table (relation), which is a set or multiset of tuples.

SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
```

```
WHERE    (Pno, Hours) IN ( SELECT    Pno, Hours
                           FROM      WORKS_ON
                           WHERE     Essn='123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value *v* (typically an attribute name) to a set or multiset *v* (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value *v* is equal to *some value* in the set *V* and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (*v* > ALL *V*) returns TRUE if the value *v* is greater than *all* the values in the set (or multiset) *V*. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT   Lname, Fname
FROM     EMPLOYEE
WHERE    Salary > ALL    ( SELECT   Salary
                           FROM     EMPLOYEE
                           WHERE    Dno=5 );
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the outer query, and another in a relation in the FROM clause of the nested query. The rule is that a reference to an unqualified attribute refers to the relation declared in the innermost nested query. For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we specify and refer to an alias (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions

Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attrib- ute of a relation declared in the outer query, the two queries are said to be correlated. We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the *outer query*. For exam- ple, we can think of Q16 as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can always be expressed as a single block query. For exam- ple,

```
SELECT      E.Fname, E.Lname
FROM EMPLOYEE AS E, DEPENDENT AS D
WHERE       E.Ssn=D.Essn AND E.Sex=D.Sex
AND E.Fname=D.Dependent_name;
```

EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.

```
SELECT      E.Fname, E.Lname FROM    EMPLOYEE AS E WHERE  EXISTS (SELECT    *
FROM DEPENDENT AS D WHERE       E.Ssn=D.Essn AND E.Sex=D.Sex
AND E.Fname=D.Dependent_name);
```

Retrieve the names of employees who have no dependents.

```
SELECT      Fname, Lname
FROM        EMPLOYEE
WHERE       NOT EXISTS ( SELECT         *
                         FROM           DEPENDENT
                         WHERE          Ssn=Essn );
```

Explicit Sets and Renaming of Attributes in SQL

We have seen several queries with a nested query in the WHERE clause. It is also pos- sible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       Pno IN (1, 2, 3);
```

- **Complex Queries**

Comparisons Involving NULLand Three-Valued Logic

SQL has various rules for dealing with NULL values. That NULL is used to represent a missing value, but that it usually has one of three different interpretations—value unknown (exists but is not known), value not available (exists but is purposely withheld), or value not applicable (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

Unknown value. A person's date of birth is not known, so it is represented by NULL in the database.

Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.

Not applicable attribute. An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used.

a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE

UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

➢ SQL Data Types

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use which are listed below −

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**char** Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	**varchar** Maximum of 8,000 characters.(Variable-length non-Unicode data).
3	**varchar(max)** Maximum length of 2E + 31 characters, Variable-length non-Unicode data (SQL Server 2005 only).

4	**text** Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**nchar** Maximum length of 4,000 characters.(Fixed length Unicode)
2	**nvarchar** Maximum length of 4,000 characters.(Variable length Unicode)
3	**nvarchar(max)** Maximum length of 2E + 31 characters (SQL Server 2005 only).(Variable length Unicode)
4	**ntext** Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types

Sr.No.	DATA TYPE & Description
1	**binary** Maximum length of 8,000 bytes(Fixed-length binary data)
2	**varbinary** Maximum length of 8,000 bytes.(Variable length binary data)
3	**varbinary(max)** Maximum length of 2E + 31 bytes (SQL Server 2005 only). (Variable length Binary data)
4	**image** Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Misc Data Types

Sr.No.	DATA TYPE & Description
1	**sql_variant** Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
2	**timestamp** Stores a database-wide unique number that gets updated every time a row gets updated
3	**uniqueidentifier** Stores a globally unique identifier (GUID)
4	**xml** Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
5	**cursor** Reference to a cursor object
6	**table** Stores a result set for later processing

➢ Schema in SQL

A Schema in SQL is a collection of database objects associated with a database. The username of a database is called a Schema owner (owner of logically grouped structures of data). Schema always belongs to a single database whereas a database can have single or multiple schemas. Also, it is also very similar to separate namespaces or containers, which stores database objects. It includes various database objects including your tables, views, procedures, index, etc

Advantages of using Schema

- You can apply security permissions for separating and protecting database objects based on user access rights.
- A logical group of database objects can be managed within a database. Schemas play an important role in allowing the database objects to be organized into these logical groups.
- The schema also helps in situations where the database object name is the same. But these objects fall under different logical groups.
- A single schema can be used in multiple databases.
- The schema also helps in adding security.
- It helps in manipulating and accessing the objects which otherwise is a complex method.
- You can also transfer the ownership of several schemas.
- The objects created in the database can be moved among schemas.

Syntax to create SQL:

```
CREATE SCHEMA [schema_ name] [AUTHORIZATION owner_name]
[DEFAULT CHARACTER SET char_set_name]
[PATH schema_name[, ...]]
[ ANSI CREATE statements [...] ]
[ ANSI GRANT statements [...] ];
```

➢ SQL Constraints

Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table. While creating a table, you can place a certain limitation on the value stored in the table. **NOT NULL:** It prevents a column from accepting null values. If you try to insert a null value in such a column, it will be rejected. It doesn’t mean a zero value.

1. **2. UNIQUE:** It ensures that values entered into a column are all different. A column with this constraint will not accept any duplicate values.
2. **3. PRIMARY KEY:** To declare a column as the primary key of the table, use the PRIMARY KEY constraint. There can be only one primary key in a table.
3. **4. CHECK:** This constraint is used to control the values entered into a field. A condition is specified along with the CHECK constraint which must be satisfied by all the values being entered into the column, otherwise, the value will be rejected.
4. **5. DEFAULT:** This constraint is used to assign default values to a column before any value is assigned to it.
5. **6. REFERENCES:** A foreign key has valued that form the primary key in another table. The two tables thus get related by using the foreign key. Columns that are chosen as a foreign key should not have values other than that present in the primary key of a related table. This

referential integrity is implemented using the REFERENCES constraint. This constraint is followed by the name of the related table and its primary key.

6. INDEX: Used to create and retrieve data from the database very quickly.

➢ Database authorization

Authorization is the process where the database manager gets information about the authenticated user. Part of that information is determining which database operations the user can perform and which data objects a user can access.

Overview of database authorization In SQL

A privilege is a type of permission for an authorization name, or a permission to perform an action or a task. The privilege allows a user to create or access database resources. Privileges are stored in the database catalogs. Authorized users can pass on privileges on their own objects to other users by using the GRANT statement. Privileges can be granted to individual users, to groups, or to PUBLIC. PUBLIC is a special group that consists of all users, including future users. Users that are members of a group will indirectly take advantage of the privileges granted to the group, where groups are supported.

A role is a database object that groups one or more privileges. Roles can be assigned to users or groups or other roles by using the GRANT statement. Users that are members of roles have the privileges that are defined for the role with which to access data.

- Administrative authority includes system-level authorization and database-level authorization:

System-level authorization

SYSADM (system administrator) authority

The SYSADM (system administrator) authority provides control over all the resources created and maintained by the database manager. The system administrator possesses all the authorities of SYSCTRL, SYSMAINT, and SYSMON authority. The user who has SYSADM authority is responsible both for controlling the database manager, and for ensuring the safety and integrity of the data.

SYSCTRL authority

The SYSCTRL authority provides control over operations that affect system resources. For example, a user with SYSCTRL authority can create, update, start, stop, or drop a database. This user can also start or stop an instance, but cannot access table data.

SYSMAINT authority

The SYSMAINT authority provides the authority required to perform maintenance operations on all databases that are associated with an instance. A user with SYSMAINT authority can update the database configuration, backup a database or table space, restore an existing database, and monitor a database. Like SYSCTRL, SYSMAINT does not provide access to table data. Users with SYSMAINT authority also have SYSMON authority.

SYSMON (system monitor) authority The SYSMON (system monitor) authority provides the authority required to use the database system monitor.

Database-level authorization

DBADM (database administrator)

The DBADM authority level provides administrative authority over a single database. This database administrator possesses the privileges required to create objects and issue database commands. The DBADM authority can be granted only by a user with SECADM authority. The DBADM authority cannot be granted to PUBLIC.

SECADM (security administrator)

The SECADM authority level provides administrative authority for security over a single database. The security administrator authority possesses the ability to manage database security objects (database roles, audit policies, trusted contexts, security label components, and security labels) and grant and revoke all database privileges and authorities.

SQLADM (SQL administrator)

The SQLADM authority level provides administrative authority to monitor and tune SQL statements within a single database. It can be granted by a user with ACCESSCTRL or SECADM authority.

WLMADM (workload management administrator)

The WLMADM authority provides administrative authority to manage workload management objects, such as service classes, work action sets, work class sets, and workloads. It can be granted by a user with ACCESSCTRL or SECADM authority. EXPLAIN (explain authority).

EXPLAIN (explain authority)

The EXPLAIN authority level provides administrative authority to explain query plans without gaining access to data. It can only be granted by a user with ACCESSCTRL or SECADM authority.

ACCESSCTRL (access control authority)

ACCESSCTRL authority can only be granted by a user with SECADM authority. The ACCESSCTRL authority cannot be granted to PUBLIC. The ACCESSCTRL authority level provides administrative authority to issue the following GRANT (and REVOKE) statements:

➢ Introduction to embedded SQL

Embedded SQL applications connect to databases and execute embedded SQL statements. The embedded SQL statements are contained in a package that must be bound to the target database server.

Embedded SQL inserts specially marked SQL statements into program source texts written in C, C++, Cobol, and other PLs. • Inside SQL statements, variables of the PL may be used where SQL allows a constant term only (parameterized queries). Insert a row into table RESULTS: EXEC SQL INSERT INTO RESULTS(SID, CAT, ENO, POINTS) VALUES (:sid, :cat, :eno, :points); . Here, sid etc. are C variables and the above may be embedded into any C source text

Embedded SQL features

Embedded SQL provides several advantages over a call-level interface:

• Embedded SQL is easy to use because it is simply Transact-SQL with some added features that facilitate using it in an application.

• It is an ANSI/ISO-standard programming language.

• It requires less coding to achieve the same results as a call-level approach.

• Embedded SQL is essentially identical across different host languages. Programming conventions and syntax change very little. Therefore, to write applications in different languages, you need not learn new syntax.

• The precompiler can optimize execution time by generating stored procedures for the Embedded SQL statements.

Creating and running an Embedded SQL program Follow these steps to create and run an Embedded SQL

Write the application program and include the Embedded SQL statements and variable declarations

- Save the application in a file with a .cp extension.
- Precompile the application. If there are no severe errors, the precompiler generates a file containing your application program. The file has the same name as the original source file,

with a different extension, depending on the requirements of your C compiler. For details, see the Open Client and Open Server Programmers Supplement for your platform.

- Compile the new source code as you would compile a standard C program.
- Link the compiled code with Client-Library
- If you specified the precompiler option to generate stored procedures, load them into Adaptive Server Enterprise by executing the generated script with isql.
- Run the application program as you would any standard C program.

Simplified Embedded SQL program

exec sql include sqlca;
main()
exec sql begin declare section; CS_CHAR user[31], passwd[31]; exec sql end declare section;
exec sql whenever sqlerror call err_p();
printf("\nplease enter your userid "); gets(user); printf("\npassword "); gets(passwd); exec sql connect :user identified by :passwd;
exec sql update titles set price = price * 1.10;
exec sql commit work;
exec sql disconnect all; }
err_p() { /* Print the error code and error message */ printf("\nError occurred: code %d.\n%s", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc); }

➢ Dynamic SQL

Dynamic SQL, an advanced methodology that lets users of Embedded SQL applications enter SQL statements while the application is running. While static SQL will suffice for most of your needs, dynamic SQL provides the flexibility to build diverse SQL statements at runtime. Dynamic SQL is a set of Embedded SQL statements that permit users of online applications to access the database interactively at application runtime.

Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

When to Use Dynamic SQL

Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at precompile time:

- text of the SQL statement (commands, clauses, and so on)
- the number of host variables
- the datatypes of host variables
- references to database objects such as columns, indexes, sequences, tables, usernames, and views

Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid SQL statement, but *not* contain the EXEC SQL clause, or the statement terminator, or any of the following embedded SQL commands:

- ALLOCATE
- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- FREE
- GET
- INCLUDE
- OPEN
- PREPARE
- SET
- WHENEVER

In most cases, the character string can contain dummy host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, Oracle makes no distinction between the following two strings:

'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'

Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections show you how to use the methods, and include sample programs that you can study.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on.

Table 13-1 Methods for Using Dynamic SQL

Method	**Kind of SQL Statement**
1	non-query without host variables
2	non-query with known number of input host variables
3	query with known number of select-list items and input host variables
4	query with unknown number of select-list items or input host variables

Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'

With Method 1, the SQL statement is parsed every time it is executed.

Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'

'DELETE FROM EMP WHERE EMPNO = :emp_number'

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are PREPARED.

Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables must be known at precompile time. For example, the following host strings qualify:

'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'

'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO =:dept_number'

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in the section "Using Method 4"). The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'

'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.

➢ PL/SQL

PL/SQL is a block structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks. Pl/SQL stands for "Procedural Language extension of SQL" that is used in Oracle. PL/SQL is integrated with Oracle database (since version 7). The functionalities of PL/SQL usually extended after each release of Oracle database. Although PL/SQL is closely integrated with SQL language, yet it adds some programming constraints that are not available in SQL.

PL/SQL Functionalities

PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers. It can support Array and handle exceptions (runtime errors). After the implementation of version 8 of Oracle database have included features associated with object orientation. You can create PL/SQL units like procedures, functions, packages, types and triggers, etc. which are stored in the database for reuse by applications.

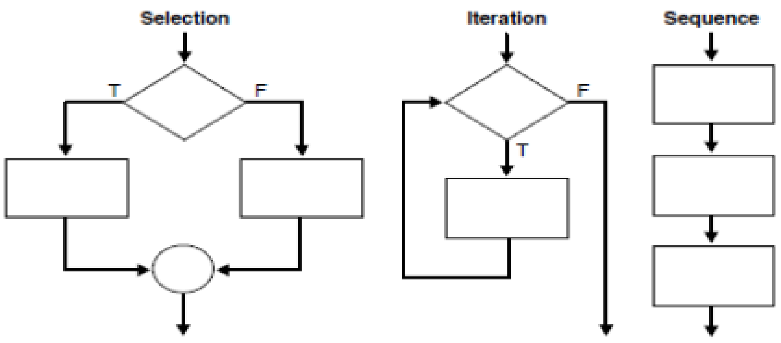
The PL/SQL is known for its combination of data manipulating power of SQL with data processing power of procedural languages. It inherits the robustness, security, and portability of the Oracle Database.

PL/SQL is not case sensitive so you are free to use lower case letters or upper case letters except within string and character literals. A line of PL/SQL text contains groups of characters known as lexical units. It can be classified as follows:

- Delimeters
- Identifiers
- Literals
- Comments

➢ Control Structures in PL/SQL

Procedural computer programs use the basic control structures.



- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.
- The sequence-structure simply executes a sequence of statements in the order in which they occur.

Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

- **Using the IF-THEN Statement**

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF) The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example: Using a Simple IF-THEN Statement

```
DECLARE
sales NUMBER(8,2) := 10100;
quota NUMBER(8,2) := 10000;
bonus NUMBER(6,2);
emp_id NUMBER(6) := 120;
BEGIN IF sales > (quota + 200) THEN bonus: = (sales - quota)/4;
UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
```

```
END IF;
END;
/
```

- **Using CASE Statements**

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
Example: Using the CASE-WHEN Statement

```
DECLARE
grade CHAR(1);
BEGIN
grade := 'B';
CASE grade
WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
END CASE;
END;
/
```

Controlling Loop Iterations: LOOP and EXIT Statements

LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

- **Using the LOOP Statement**

The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an EXIT statement to stop looping and prevent an infinite loop. You can place one or more EXIT statements anywhere inside a loop, but not outside a loop. There are two forms of EXIT and EXIT-WHEN.

- **Using the EXIT Statement**

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

- **Using the EXIT-WHEN Statement**

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

- **Labeling a PL/SQL Loop**

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

- **Using the WHILE-LOOP Statement**

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
sequence_of_statements
END LOOP;
```

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once

Example: Using a Simple FOR LOOP Statement

```
DECLARE
p NUMBER := 0;
BEGIN
FOR k IN 1..500 LOOP -- calculate pi with 500 terms
p := p + ( ( (-1) ** (k + 1) ) / ((2 * k) - 1) );
END LOOP;
p := 4 * p;
DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); -- print result
END;
/
```

Sequential Control: GOTO and NULL Statements

The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.
Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

- **Using the GOTO Statement**

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements

Example: Using a Simple GOTO Statement

```
DECLARE
p VARCHAR2(30);
n PLS_INTEGER := 37; -- test any integer > 2 for prime
BEGIN
FOR j in 2..ROUND(SQRT(n)) LOOP
IF n MOD j = 0 THEN -- test for prime
p := ' is not a prime number'; -- not a prime number
GOTO print_now;
END IF;
END LOOP;
p := ' is a prime number';
<<print_now>>
DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

1. Using the NULL Statement

The NULL statement does nothing and passes control to the next statement. Some languages refer to such instruction as a no-op (no operation).

➢ PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

1. **CREATE** [OR REPLACE] **FUNCTION** function_name [parameters]
2. [(parameter_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6. < function_body >
7. **END** [function_name];

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that 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.

PL/SQL Function Example

Let's see a simple example to **create a function**.

1. **create** or replace **function** adder(n1 in number, n2 in number)
2. **return** number
3. **is**
4. n3 number(8);
5. **begin**
6. n3 :=n1+n2;
7. **return** n3;
8. **end**;
9. /

Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

```
DECLARE
  num number;
  factorial number;
 FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
     f := 1;
  ELSE
     f := x * fact(x-1);
END IF;
RETURN f;
END;
 BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

PL/SQL Drop Function

Syntax for removing your created function:

If you want to remove your created function from the database, you should use the following syntax.

DROP FUNCTION function_name;

➢ PL/SQL Cursor

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit Cursors
- Explicit Cursors

1) PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

For example: When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a

SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table specifies the status of the cursor with each of its attribute.

Attribute	**Description**
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	**NAME**	**AGE**	**ADDRESS**	**SALARY**
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

Create procedure:

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output:

```
  6 customers updated
  PL/SQL procedure successfully completed.
```

Now, if you check the records in customer table, you will find that the rows are updated.

1. **select** * **from** customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

2) PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors 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.

Following is the syntax to create an explicit cursor:

Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

1. **CURSOR** cursor_name **IS** select_statement;;

Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

1) Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor decleration

1. **CURSOR name IS**
2. **SELECT** statement;

2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open:

1. **OPEN** cursor_name;

3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

Syntax for cursor fetch:

1. **FETCH** cursor_name **INTO** variable_list;

4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax for cursor close:

1. **Close** cursor_name;

PL/SQL Explicit Cursor Example

Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create procedure:

Execute the following program to retrieve the customer name and address.

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%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;
/
```

Output:

```
1  Ramesh Allahabad
2 Suresh Kanpur
3 Mahesh Ghaziabad
4 Chandan Noida
5 Alex  Paris
6 Sunita Delhi
PL/SQL procedure successfully completed.
```

➢ PL/SQL Exception Handling

What is Exception

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

- System-defined Exceptions
- User-defined Exceptions

PL/SQL Exception Handling

Syntax for exception handling:y Catch

Following is a general syntax for exception handling:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  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 of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT* FROM COUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

```
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;
/
```

After the execution of above code at SQL Prompt, it produces the following result:

```
No such customer!
PL/SQL procedure successfully completed.
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

```
DECLARE
  c_id customers.id%type := 5;
  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;
/
```

After the execution of above code at SQL prompt, you will get the following result:

```
Name: alex
Address: paris
PL/SQL procedure successfully completed.
```

Raising Exceptions

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

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;
```

PL/SQL User-defined Exceptions

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure

DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Syntax for user define exceptions

DECLARE

my-exception EXCEPTION;

PL/SQL Pre-defined Exceptions

There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.

For example: NO_DATA_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions:

Exception	**Oracle Error**	**SQL Code**	**Description**
ACCESS_INTO_NULL	06530	-6530	It is raised when a NULL object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than exists to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when s program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a select into statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.

SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

➢ PL/SQL Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.
Triggers are stored programs, which are automatically executed or fired when some event occurs.
Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATES, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating a trigger:

Syntax for creating trigger:

CREATE [OR REPLACE] **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;
```

Here,

- CREATE [OR REPLACE] TRIGGER trigger_name: It creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would 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 would 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, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

Create table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create trigger:

Let's take a program to create 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;
/
```

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output:

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
6 customers updated
```

➢ PL/SQL Packages

A package is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification (spec) and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms.
To create package specs, use the SQL statement CREATE PACKAGE. A CREATE PACKAGE BODY statement defines the package body.

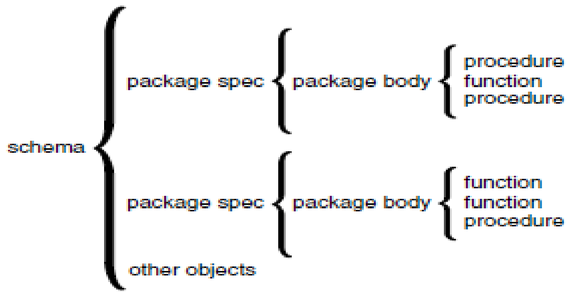
What Goes In a PL/SQL Package?

The following is contained in a PL/SQL package:

- Get and Set methods for the package variables, if you want to avoid letting other procedures read and write them directly.
- Cursor declarations with the text of SQL queries. Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you need to change a query that is used in many places.
- Declarations for exceptions. Typically, you need to be able to reference these from different procedures, so that you can handle exceptions within called subprograms.
- Declarations for procedures and functions that call each other. You do not need to worry about compilation order for packaged procedures and functions, making them more convenient than standalone stored procedures and functions when they call back and forth to each other.
- Declarations for overloaded procedures and functions. You can create multiple variations of a procedure or function, using the same names but different sets of parameters.
- Variables that you want to remain available between procedure calls in the same session. You can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored procedures or functions, you must declare the type in a package so that both the calling and called subprogram can refer to it.

Advantages of PL/SQL Packages

- Modularity
- Easier Application Design
- Information Hiding
- Added Functionality
- Better Performance



Example: A Simple Package Specification Without a Body

```
CREATE PACKAGE trans_data AS -- bodiless package
TYPE TimeRec IS RECORD (
minutes SMALLINT,
hours SMALLINT);
TYPE TransRec IS RECORD (
category VARCHAR2(10),
account INT,
amount REAL,
time_of TimeRec);
minimum_balance CONSTANT REAL := 10.00;
number_processed INT;
insufficient_funds EXCEPTION;
END trans_data;
/
```

UNIT-III

➢ Object

An object database is managed by an object-oriented database management system (OODBMS). The database combines object-oriented programming concepts with relational database principles.
Objects are the basic building block and an instance of a class, where the type is either built-in or user-defined.

➢ XML

XML is widely accepted as a new standard for documents having structural information on the Web. Typically, the number of Web documents is very large and, therefore, storing and managing them require an efficient storage manager. There exist several types of XML storage systems, but most of them use relational DBMSs, and this is what currently available commercial DBMSs do as well. Naturally, their focus has been on storing XML documents as relational database records.

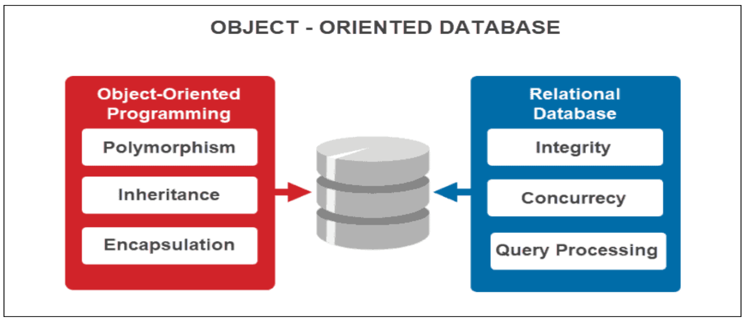
Storing XML documents as database records requires a specification of the mapping from the document structures to database schema. Currently, most commercial DBMSs provide such specification languages, but the languages are proprietary and limited to specifying a mapping to relational databases only. This limitation keeps us from exploiting the powerful modeling capabilities of object-oriented/object-relational (OO/OR) DBMSs like the references and the collections. Furthermore, learning the proprietary languages is a burden to users.

we propose an XML storage system geared for an OO/OR DBMS. For this purpose, we first analyze the mapping from XML document structures to OO/OR schema and propose a specification language based on the standard XML Schema Language. Then, we propose a set of system catalog classes for storing user-specified mapping information in the database and propose a detailed algorithm for storing XML documents in an OO/OR database based on the mapping information.

Object-oriented databases are a type of database management system. Different database management systems provide additional functionalities. Object-oriented databases add the database functionality to object programming languages, creating more manageable code bases.

Object Database Definition

An object database is managed by an object-oriented database management system (OODBMS). The database combines object-oriented programming concepts with relational database principles.
Objects are the basic building block and an instance of a class, where the type is either built-in or user-defined.



Classes provide a schema or blueprint for objects, defining the behavior.
Methods determine the behavior of a class.
Pointers help access elements of an object database and establish relations between objects.
The main characteristic of objects in OODBMS is the possibility of user-constructed types. An object created in a project or application saves into a database as is.
Object-oriented databases directly deal with data as complete objects. All the information comes in one instantly available object package instead of multiple tables.

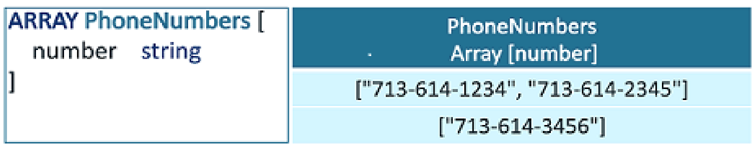
➢ Complex Data types

Complex types are nested data structures composed of primitive data types. These data structures can also be composed of other complex types. Some examples of complex types include struct(row), array/list, map and union.

Complex types are supported by most programming languages including Python, C++ and Java. They are also supported by databases like PostgreSQL which introduced composite (struct) type And, they are supported in Vertica. What we are adding is a richer, more complete support for these types, not just querying complex types in ORC and Parquet, but doing so rapidly, without converting the file to a far larger row and column format first. We are even adding the capability to store data in efficient complex types in Vertica's own ROS storage for even faster processing.

Examples of Complex Types

Array – Array is a homogenous collection of elements of the same data type. These can be int, float, or any kind of string (string, char, varchar, etc.) or another complex type. The Array data type is quite similar to Java's arraylist which is a strongly typed list but also lets you nest the lists to create arrays.

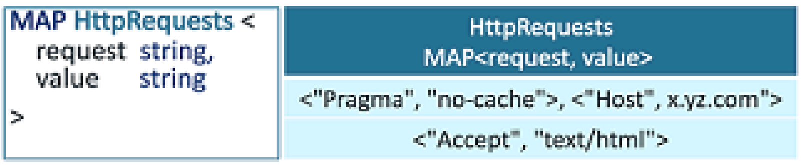


```
ARRAY PhoneNumbers [
   number   string
]
```

PhoneNumbers Array [number]
["713-614-1234", "713-614-2345"]
["713-614-3456"]

Example of an array could be PhoneNumbers where each number is defined as a string data type. All elements in this array would have the same data type, ie. string.

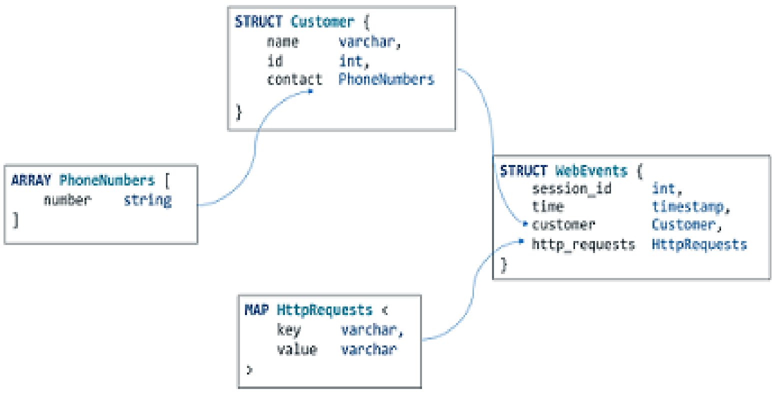
Map – Map data types are key-value pairs. This type associates a key with a value. For example, a city could be associated with a zipcode, or an address with a phone number.



```
MAP HttpRequests <
   request  string,
   value    string
>
```

HttpRequests MAP<request, value>
<"Pragma", "no-cache">, <"Host", x.yz.com">
<"Accept", "text/html">

Nested Complex Types



```
STRUCT Customer {
    name      varchar,
    id        int,
    contact   PhoneNumbers
}

ARRAY PhoneNumbers [
    number    string
]

STRUCT WebEvents {
    session_id      int,
    time            timestamp,
    customer        Customer,
    http_requests   HttpRequests
}

MAP HttpRequests <
    key      varchar,
    value    varchar
>
```

➢ **Structured types**

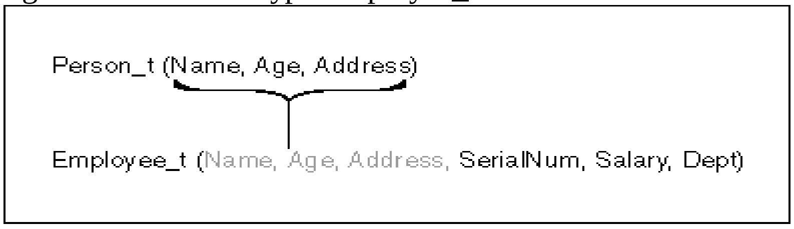
A structured type is a user-defined data type containing one or more named attributes, each of which has a data type. Attributes are properties that describe an instance of a type.

A geometric shape, for example, might have attributes such as its list of Cartesian coordinates. A person might have attributes of name, address, and so on. A department might have attributes of a name or some other kind of ID.

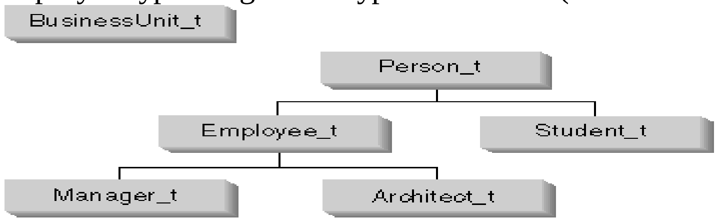
A structured type also includes a set of method specifications. Methods enable you to define behaviors for structured types. Like user-defined functions (UDFs), methods are routines that extend SQL. In the case of methods, however, the behavior is integrated solely with a particular structured type.

A structured type can be used as the type of a table, view, or column. When used as the type for a table, that table is known as a typed table and when used as the type for a view, that view is known as a typed view. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of the typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type.

A type cannot be dropped when certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes a direct or indirect use of
Creating structured types
This topic describes how to create structured types. A structured type is a user-defined type that contains one or more attributes, each of which has a name and a data type of its own. A structured type can serve as the type of a table or view in which each column of the table derives its name and data type from one of the attributes of the structured type. A structured type can also serve as a type of a column or a type for an argument to a routine. the type
Structured type hierarchies it is certainly possible to model objects such as people using traditional relational tables and columns. However, structured types offer an additional property of inheritance. That is, a structured type can have subtypes that reuse all of its attributes and contain additional attributes specific to the subtype. The original type is the supertype. For example, the structured type Person_t might contain attributes for Name, Age, and Address. A subtype of Person_t might be Employee_t that contains all of the attributes Name, Age, and Address and, in addition, contains attributes for SerialNum, Salary, and BusinessUnit.
Figure 1. Structured type Employee_t inherits attributes from supertype Person_t


Person_t (Name, Age, Address)
Employee_t (Name, Age, Address, SerialNum, Salary, Dept)

A set of subtypes based (at some level) on the same supertype is known as a type hierarchy. For example, a data model might need to represent a special type of employee called a manager. Managers have more attributes than employees who are not managers. The Manager_t type inherits the attributes defined for an employee, but also is defined with some additional attributes of its own, such as a special bonus attribute that is only available to managers.
The following figure presents an illustration of the various subtypes that might be derived from person and employee types: Figure 2. Type hierarchies (BusinessUnit_t and Person_t)



In Figure 2, the person type Person_t is the root type of the hierarchy. Person_t is also the supertype of the types below it--in this case, the type named Employee_t and the type named Student_t. The relationships among subtypes and supertypes are transitive; this means that the relationship between subtype and supertype exists throughout the entire type hierarchy. So, Person_t is also a supertype of types Manager_t and Architect_t.

➢ Inheritance in SQL

Inheritance is the mechanism that connects subtypes in a hierarchy to their supertypes.
Subtypes automatically inherit the attributes and methods of their parent type. Also, the inheritance link remains alive. Subtypes automatically acquire any changes made to these attributes or methods in the parent: any attributes or methods updated in a supertype are updated in subtypes as well.
Note:

Oracle only supports single inheritance. Therefore, a subtype can derive directly from only one supertype, not more than one.

With object types in a type hierarchy, you can model an entity such as a customer, and also define different specializing subtypes of customers under the original type. You can then perform operations on a hierarchy and have each type implement and execute the operation in a special way.

Let's create two tables: A table cities and table capitals. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities.

```
CREATE TABLE capitals ( name text ,  population real,  altitude   int, -- (in ft)  state  char(2));
CREATE TABLE non_capitals (name text, population real, and altitude int -- (in ft));
CREATE VIEW cities AS    SELECT name, population, altitude FROM capitals
   UNION
  SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, to name one thing. A better solution is this: CREATE TABLE cities (name text, population real, and altitude int - (in ft));

```
CREATE TABLE capitals ( state char(2)) INHERITS (cities);
```

In this case, a row of capitals inherits all columns (name, population, and altitude) from its parent, cities. The type of the column name is text, a native PostgreSQL type for variable length character strings. State capitals have an extra column, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 ft.:

```
SELECT name, altitude   FROM cities WHERE altitude > 500;
```

Which returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 ft. or higher:

```
SELECT name, altitude
   FROM ONLY cities
   WHERE altitude > 500;
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Here the ONLY before cities indicates that the query should be run over only the cities table, and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- SELECT, UPDATE, and DELETE -- support this ONLY notation.

➢ Arrays in SQL statements

SQL introduces limited support for a single collection type known as arrays. Arrays are variable-sized ordered collections with a declared maximum cardinality. Here's an example of how to declare a column with array type:

```
CREATE TABLE genome_sequences (
   sequence_id BIGINT NOT NULL PRIMARY KEY,
   chromosome number INT NOT NULL,    start offset BIGINT NOT NULL,
   codons CHAR (3) ARRAY [1000] NOT NULL);
```

Array values are created with the ARRAY constructor:

```
INSERT INTO genome_sequences VALUES (
   10032423, 3, 432432,
   ARRAY['CUG', 'AAG', 'GGU', 'ACU', 'CUU', 'GGU', 'UGG', 'UAA']);
```

The length of an array is retrieved with the CARDINALITY function:

```
SELECT sequence_id, CARDINALITY (codons) as sequence_length
FROM genome_sequences
ORDER BY sequence_length;
```

And the actual values can be selected with the usual bracket syntax:

```
SELECT
   sequence_id,    codons [1] as first_codon,    codons [CARDINALITY (codons)] as last_codon
FROM genome_sequences ORDER BY sequence_id;
```

Arrays can be concatenated with the usual notation:

```
CREATE VIEW spliced_sequences AS
SELECT s1.sequence_id, s1.codons || s2.codons AS spliced_codons
FROM genome_sequences s1, genome_sequences s2
WHERE s1.chromosome_number = s2.chromosome_number
AND s2.start_offset = s1.start_offset + CARDINALITY (s1.codons);
```

Array elements can be set individually in the SET clause of an UPDATE statement, or the entire **array can be set as the target of an INSERT or UPDATE:**

```
UPDATE genome_sequences
SET codons =     (SELECT codons     FROM spliced_sequences
    WHERE spliced_sequences.sequence_id = genome_sequences.sequence_id);
```

Array assignment in which the maximum size of the target is less than the actual size of a source value results in an exception. In the example above, this will occur if two sequences spliced together make up a sequence longer than 1000 codons.

Arrays can be compared for exact match with = or <>:

```
CREATE VIEW matching_sequences AS
SELECT s1.sequence_id as sid1, s2.sequence_id as sid2
FROM genome_sequences s1, genome_sequences s2
WHERE s1.codons = s2.codons;
```

➢ Multiset Types

A multiset is much like an array, but unordered, and has more useful operators. Although multisets and arrays are both collections, they don't mix (e.g. you can't INSERT a multiset into an array column).

Multiset constructors are similar to array constructors:

```
CREATE TABLE logins(
   session_id INT NOT NULL PRIMARY KEY,
   successful BOOLEAN NOT NULL,    uid INT,
   Attempts ROW(VARCHAR (128),VARCHAR(128)) MULTISET);
INSERT INTO logins VALUES (1000, true,0, MULTISET (
      ROW('root','31337'),
      ROW ('scott','tiger'),
      ROW ('root','beer')));
INSERT INTO logins VALUES (1001, false, 0, MULTISET (SELECT ROW (name, password) FROM bogus_accounts));
```

Elements of the array may be declared with almost any datatype

CARDINALITY returns the total number of elements in a multiset (not the number of distinct elements). Multisets cannot be concatenated (since they aren't ordered); instead, a number of set operators are provided for multisets:

```
multiset1 MULTISET UNION [ALL|DISTINCT] multiset2
multiset1 MULTISET INTERSECT [ALL|DISTINCT] multiset2
multiset1 MULTISET EXCEPT [ALL|DISTINCT] multiset2
```

UNNEST can be used on a multiset (but not WITH ORDINALITY, obviously). The multiset-specific ELEMENT operator converts a multiset with cardinality 1 into a row expression aggregation operators are provided for collecting row values into multisets, and for aggregating multisets.

➢ Object identity

Object identity is a fundamental object orientation concept. With object identity, objects can contain or refer to other objects. Identity is a property of an object that distinguishes the object from all other objects in the application.

There are many techniques for identifying objects in programming languages, databases and operating systems. According to the authors the most commonly used technique for identifying objects is user-defined names for objects. There are of course practical limitations to the use of variable names without the support of object identity.

To identify an object with a unique key (also called identifier keys) is a method that is commonly used in database management systems. Using identifier keys for object identity confuses identity and data values. According to the authors there are three main problems with this approach:

1. Modifying identifier keys. Identifier keys cannot be allowed to change, even though they are user-defined descriptive data
2. Non-uniformity. The main source of non-uniformity is that identifier keys in different tables have different types or different combinations of attributes. And more serious problem is that the attribute(s) to use for an identifier key may need to change
3. "Unnatural" joins. The use of identifier keys causes joins to be used in retrievals instead of simpler and more direct object retrievals.

There are many operations associated with identity. Since the state of an object is different from its identity, there are three types of equality predicates from comparing objects. The most obvious equality predicate is the identical predicate, which checks whether two objects are actually one and the same object. The two other equality predicates (shallow equal and deep equal) actually compare the states of objects. Shallow equal goes one level deep in comparing corresponding instances variables. Deep equal ignores identities and compares the values of corresponding base objects. As far as copying objects, the counterparts of deep and shallow equality are deep and shallow copy.

➢ Reference types

For every structured type you create, Db2® automatically creates a companion type. The companion type is called *a* reference type and the structured type to which it refers is called a referenced type. Typed tables can make special use of the reference type. You can also use reference types in SQL statements in the same way that you use other user-defined types. To use a reference type in an SQL statement, use REF represents the referenced type.

Db2 uses the reference type as the type of the object identifier column in typed tables. The object identifier uniquely identifies a row object in the typed table hierarchy. Db2 also uses reference types to store references to rows in typed tables. You can use reference types to refer to each row object in the table.

References are strongly typed. Therefore, you must have a way to use the type in expressions. When you create the root type of a type hierarchy, you can specify the base type for a reference with the REF USING clause of the CREATE TYPE statement. The base type for a reference is called the representation type. If you do not specify the representation type with the REF USING clause, Db2 uses the default data type of VARCHAR(16) FOR BIT DATA. The representation type of the root type is inherited by all its subtypes. The REF USING clause is only valid when you define the root type of a hierarchy. In the examples used throughout this section, the representation type for the BusinessUnit_t type is INTEGER, while the representation type for Person_t is VARCHAR (13).

➢ Persistent Programming Languages

Persistence is a property of data values which allows them to endure for an arbitrary time. For example, heap technology is introduced into programming languages, to extend the persistence of data from the activation period of a block to the execution time of a program. This is still not full persistence since there is an upper bound (the execution time) to the longevity of data. It is as important that brief lifetimes (transience) should be included in persistence otherwise a programmer has difficulty with intermediate results. We identify three principles which direct the provision of persistence:

i) persistence independence the persistence of a data object is independent of how the program manipulates that data object, and conversely, a fragment of program is expressed independently of the persistence of the data it manipulates;
ii) persistent data type orthoganality: consistent with the general programming language design principle of data type completeness, all data objects, whatever their type, should be allowed the full range of persistence;
iii) orthogonal persistence management: the choice of how to provide and identify persistence is orthogonal to the choice of type system, computational model and control structures of the language.

Motivation for persistent programming

The initial motivation arose from the difficulties of storing and restoring data structures arising in CAD/CAM research.

1) Contorted mappings were needed to store arrays and graphs represented as references and records onto database structures. These contorted mappings had a number of costs:

i) they introduced concepts extraneous to the computation, obscuring code and confusing programmers;

ii) they did not precisely preserve information and were not subject to type checking, consequently they were a source of errors;

iii)it was difficult to incrementally translate in both directions and consequently more data than necessary was loaded and unloaded per program run;

iv) there were large computational overheads performing the translations; and v) concurrent use of the data was excluded.

➢ OO VS OR

The object-oriented approach deals with data at a much higher level than a relational database. Whereas a relational database deals with data at the level of columns and rows, an object-oriented system deals with objects, which may be any number of collections of data items. An object may be an order, an inventory list, or any real-world representation of a physical object. For example, consider an object called ORDER. ORDER is a logical object to the object-oriented system, and each ORDER will have associated data items and behaviors. Behaviors might include PLACE_ORDER, CHANGE_ORDER, and so on.

At the relational database level, an ORDER is really a consolidation of many different columns from many different tables. The customer name comes from the CUSTOMER table; order date comes from the ORDER table, quantity from the LINE_ITEM table and item description from the ITEM table. Hence, a single behavior for an object may cause changes to many tables within the relational database.

- **Implementing OR Features**

  The relational model with object database enhancements is sometimes referred to as the **object-relational model**. We introduced SQL as the standard language for RDBMS

  1. User-Defined Types and Complex Structures for Objects

  To allow the creation of complex-structured objects, and to separate the declaration of a type from the creation of a table, SQL now provides **user-defined types** (**UDT**s). In addition, four collection types have been included to allow for multivalued types and attributes in order to specify complex-structured objects rather than just simple (flat) records. The user will create the UDTs for a particular application as part of the database schema. A **UDT** may be specified in its simplest form using the following syntax:

  CREATE TYPE TYPE_NAME AS (<component declarations>);

An Object relational model is a combination of a Object oriented database model and a Relational database model. So, it supports objects, classes, inheritance etc. just like Object Oriented models and has support for data types, tabular structures etc. like Relational data model. One of the major goals of Object relational data model is to close the gap between relational databases and the object oriented practices frequently used in many programming languages such as C++, C#, Java etc.

History of Object Relational Data Model

Both Relational data models and Object oriented data models are very useful. But it was felt that they both were lacking in some characteristics and so work was started to build a model that was a combination of them both. Hence, Object relational data model was created as a result of research that was carried out in the 1990's.

Advantages of Object Relational model

The advantages of the Object Relational model are −

Inheritance

The Object Relational data model allows its users to inherit objects, tables etc. so that they can extend their functionality. Inherited objects contains new attributes as well as the attributes that were inherited.

Complex Data Types

Complex data types can be formed using existing data types. This is useful in Object relational data model as complex data types allow better manipulation of the data.

Extensibility

The functionality of the system can be extended in Object relational data model. This can be achieved using complex data types as well as advanced concepts of object oriented model such as inheritance.

1. **Triggers:**Triggers are actions performed when certain conditions are met on the data.
   A trigger contains of three parts.
   - **(i). event –** The change in the database that activates the trigger is event.

- **(ii). condition –** A query or test that is run when the trigger is activated.
- **(iii). action –** A procedure that is executed when trigger is activated and the condition met is true.

2. **Client server execution and remote database access**
   Client server technology maintains a many to one relationship of clients(many) and server(one). We have commands in SQL that control how a client application can access the database over a network.
3. **Security and authentication**
   SQL provides a mechanism to control the database meaning it makes sure that only the particular details of the database is to be shown the user and the original database is secured by DBMS.
4. **Embedded SQL**
   SQL provides the feature of embedding host languages such as C, COBOL, Java for query from their language at runtime.
5. **Transaction Control Language**
   Transactions are an important element of DBMS and to control the transactions, TCL is used which has commands like commit, rollback and savepoint.
   commit: It saves the database at any point whenever database is consistent
   Syntax:commit;
   rollback: It rollbacks/undo to the previous point of the transaction
   Syntax:rollback;
   savepoint: It goes back to the previous transaction without going back to the entire transaction. **Syntax:**savepoint;
6. **Advanced SQL**
   The current features include OOP ones like recursive queries, decision supporting queries and also query supporting areas like data mining, spatial data and XML(Xtensible Markup Language)

Disadvantages of Object Relational model
The object relational data model can get quite complicated and difficult to handle at times as it is a combination of the Object oriented data model and Relational data model and utilizes the functionalities of both of them.

- **XML stands for Extensible Markup Language**. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.
There are three important characteristics of XML that make it useful in a variety of systems and solutions

- **XML is extensible** − XML allows you to create your own self-descriptive tags, or language, that suits your application.
- **XML carries the data, does not present it** − XML allows you to store the data irrespective of how it will be presented.
- **XML is a public standard** − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML Usage
A short list of XML usage says it all −

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange the information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange the data, which can customize your data handling needs.
- XML can easily be merged with style sheets to create almost any desired output.

- Virtually, any type of data can be expressed as an XML document.

➢ XML STRUCTURE

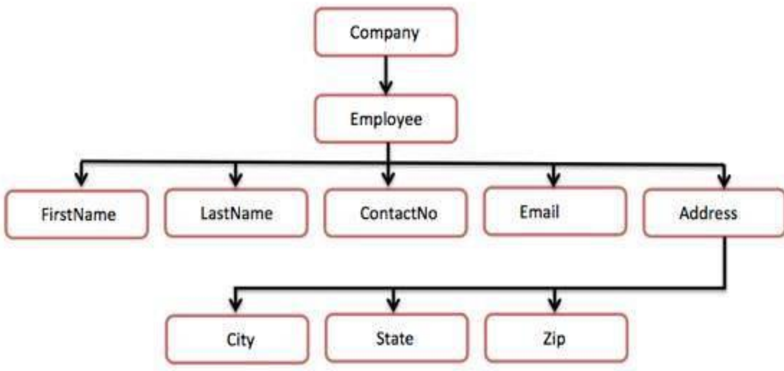
An XML document is always descriptive. The tree structure is often referred to as **XML Tree** and plays an important role to describe any XML document easily.

The tree structure contains root (parent) elements, child elements and so on. By using tree structure, you can get to know all succeeding branches and sub-branches starting from the root. The parsing starts at the root, then moves down the first branch to an element, take the first branch from there, and so on to the leaf nodes.

Example

Following example demonstrates simple XML tree structure −

```
<? xml version = "1.0"?>
<Company>
  <Employee>
    <FirstName>Tanmay</FirstName>
    <LastName>Patil</LastName>
    <ContactNo>1234567890</ContactNo>
    <Email>tanmaypatil@xyz.com</Email>
    <Address>
      <City>Bangalore</City>
      <State>Karnataka</State>
      <Zip>560212</Zip>
    </Address>
  </Employee>
</Company>
```



Following tree structure represents the above XML document −

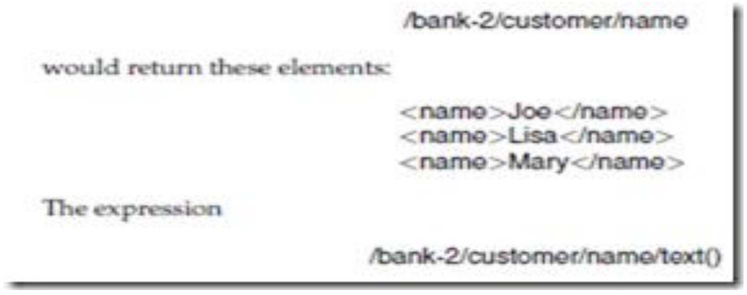
In the above diagram, there is a root element named as <company>. Inside that, there is one more element <Employee>. Inside the employee element, there are five branches named <FirstName>, <LastName>, <ContactNo>, <Email>, and <Address>. Inside the <Address> element, there are three sub-branches, named <City> <State> and <Zip>.

➢ Querying and Transformation

In particular, tools for querying and transformation of XML data are essential to extract information from large bodies of XML data, and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined into a single tool.

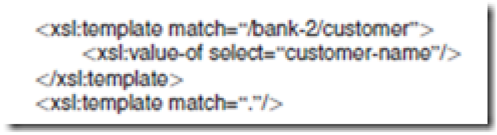
Several languages provide increasing degrees of querying and transformation capabilities:

• XPath is a language for path expressions, and is actually a building block for the remaining two query languages. XPath addresses parts of an XML document by means of path expressions. The language can be viewed as an extension of the simple path expressions in object-oriented and object-relational databases A **path expression** in XPath is a sequence of location steps separated by “/” (in- stead of the “.” operator that separates steps in SQL:1999). The result of a path ex- pression is a set of values. For instance, on the document in Figure 10.8, the XPath expression



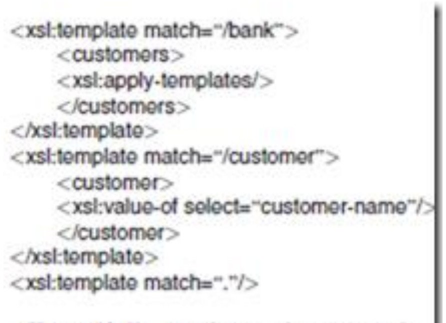
```
                    /bank-2/customer/name
would return these elements:
                    <name>Joe</name>
                    <name>Lisa</name>
                    <name>Mary</name>
The expression
                    /bank-2/customer/name/text()
```

• XSLT was designed to be a transformation language, as part of the XSL style sheet system, which is used to control the formatting of XML data into HTML or other print or display languages. Although designed for formatting, XSLT can generate XML as output, and can express many interesting queries. Furthermore, it is currently the most widely available language for manipulating XML data.
XSLT transformations are expressed as a series of recursive rules, called **templates**.
In their basic form, templates allow selection of nodes in an XML tree by an XPath expression. However, templates can also generate new XML content, so that selection and content generation can be mixed in natural and powerful ways. While XSLT can be used as a query language, its syntax and semantics are quite dissimilar from those of SQL.
A simple template for XSLT consists of a **match** part and a **select** part. Consider this XSLT code:



```
<xsl:template match="/bank-2/customer">
    <xsl:value-of select="customer-name"/>
</xsl:template>
<xsl:template match="."/>
```

The xsl:template match statement contains an XPath expression that selects one or more nodes. The first template matches customer elements that occur as children of the bank-2 root element. The xsl:value-of statement enclosed in the match statement outputs values from the nodes in the result of the XPath expression. The first template outputs the value of the customer-name subelement; note that the value does not contain the element tag.
Note that the second template matches all nodes. This is required because the de- fault behavior of XSLT on subtrees of the input document that do not match any template is to copy the subtrees to the output document.
XSLT copies any tag that is not in the xsl namespace unchanged to the output. Figure 10.10 shows how to use this feature to make each customer name from our example appear as a subelement of a "<customer>" element, by placing the xsl:value-of statement between <customer> and </customer>.



```
<xsl:template match="/bank">
    <customers>
    <xsl:apply-templates/>
    </customers>
</xsl:template>
<xsl:template match="/customer">
    <customer>
    <xsl:value-of select="customer-name"/>
    </customer>
</xsl:template>
<xsl:template match="."/>
```

Figure 10.11 Applying rules recursively.

Structural recursion is a key part of XSLT. Recall that elements and subelements naturally form a tree structure. The idea of structural recursion is this: When a template matches an element in the tree structure, XSLT can use structural recursion to apply template rules recursively on subtrees, instead of just outputting a value. It applies rules recursively by the xsl:apply-templates directive, which appears inside other templates.

For example, the results of our previous query can be placed in a surrounding <customers> element by the addition of a rule using xsl: apply-templates, as in Figure 10.11 The new rule matches the outer "bank" tag, and constructs a result document by applying all other templates to the subtrees appearing within the bank element, but wrapping the results in the given <customers> </customers> element. Without recursion forced by the <xsl:apply-templates/> clause, the template would output <customers> </customers>, and then apply the other templates on the subelements.

• XQuery has been proposed as a standard for querying of XML data. XQuery combines features from many of the earlier proposals for querying XML, in particular the language Quilt.

XML DOCUMENT SCHEMA

XML Schema is commonly known as **XML Schema Definition (XSD)**. It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

Syntax **You need to declare a schema in your XML document as follows −**

Example

The following example shows how to use schema −

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "contact">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "name" type = "xs:string" />
            <xs:element name = "company" type = "xs:string" />
            <xs:element name = "phone" type = "xs:int" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

Elements

As we saw in the XML - Elements chapter, elements are the building blocks of XML document. An element can be defined within an XSD as follows −

```
<xs:element name = "x" type = "y"/>
```

Definition Types You can define XML schema elements in the following ways −

Simple Type Simple type element is used only in the context of the text. Some of the predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example −

```
<xs:element name = "phone_number" type = "xs:int" />
```

Complex Type

A complex type is a container for other element definitions. This allows you to specify which child elements an element can contain and to provide some structure within your XML documents. For example

```
<xs:element name = "Address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "name" type = "xs:string" />
      <xs:element name = "company" type = "xs:string" />
      <xs:element name = "phone" type = "xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In the above example, *Address* element consists of child elements. This is a container for other **<xs:element>** definitions, that allows to build a simple hierarchy of elements in the XML document.

Global Types

With the global type, you can define a single type in your document, which can be used by all other references. For example, suppose you want to generalize the *person* and *company* for different addresses of the company. In such case, you can define a general type as follows −

```
<xs:element name = "AddressType">
  <xs:complexType>
    <xs:sequence>
      <xs: element name = "name" type = "xs:string" />
      <xs: element name = "company" type = "xs:string" />
    </xs: sequence>
  </xs:complexType>
</xs:element>
```

Now let us use this type in our example as follows −

```
<xs:element name = "Address1">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "address" type = "AddressType" />
      <xs: element name = "phone1" type = "xs:int" />
    </xs:sequence>
  </xs: complexType>
</xs:element>
<xs:element name = "Address2">
  <xs:complexType>
    <xs:sequence>
      <xs: element name = "address" type = "AddressType" />
      <xs: element name = "phone2" type = "xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Instead of having to define the name and the company twice (once for *Address1* and once for *Address2*), we now have a single definition. This makes maintenance simpler, i.e., if you decide to add "Postcode" elements to the address, you need to add them at just one place.

Attributes

Attributes in XSD provide extra information within an element. Attributes have *name* and *type* property as shown below −

<xs: attribute name = "x" type = "y"/>

- **API inXML**

Simple API for XML (SAX), an event-driven, serial-access mechanism for accessing XML documents. This is the protocol that most servlets and network-oriented programs will want to use to transmit and receive XML documents, because it's the fastest and least memory-intensive mechanism that is currently available for dealing with XML documents.

The SAX protocol requires a lot more programming than the Document Object Model (DOM). It's an event-driven model (you provide the callback methods, and the parser invokes them as it reads the XML data), which makes it harder to visualize. Finally, you can't "back up" to an earlier part of the document, or rearrange it, any more than you can back up a serial data stream or rearrange characters you have read from that stream.

For those reasons, developers who are writing a user-oriented application that displays an XML document and possibly modifies it will want to use the DOM mechanism described in the next part of the tutorial, Document Object Model.

However, even if you plan to build with DOM apps exclusively, there are several important reasons for familiarizing yourself with the SAX model:

- Same Error Handling
  When parsing a document for a DOM, the same kinds of exceptions are generated, so the error handling for JAXP SAX and DOM applications are identical.
- Handling Validation Errors
  By default, the specifications require that validation errors (which you'll be learning more about in this part of the tutorial) are ignored. If you want to throw an exception in the event of a validation error (and you probably do) then you need to understand how the SAX error handling works.
- Converting Existing Data
  As you'll see in the DOM section of the tutorial, there is a mechanism you can use to convert an existing data set to XML--however, taking advantage of that mechanism requires an understanding of the SAX model.

- **Applications of XML**

XML has a variety of uses for Web, e-business, and portable applications.

The following are some of the many applications for which XML is useful:

Web publishing: XML allows you to create interactive pages, allows the customer to customize those pages, and makes creating e-commerce applications more intuitive. With XML, you store the data once and then render that content for different viewers or devices based on style sheet processing using an Extensible Style Language (XSL)/XSL Transformation (XSLT) processor.

Web searching and automating Web tasks: XML defines the type of information contained in a document, making it easier to return useful results when searching the Web:For example, using HTML to search for books authored by Tom Brown is likely to return instances of the term 'brown' outside of the context of author. Using XML restricts the search to the correct context (for example, the information contained in the <author> tag) and returns only the information that you want. By using XML, Web agents and robots (programs that automate Web searches or other tasks) are more efficient and produce more useful results.

General applications: XML provides a standard method to access information, making it easier for applications and devices of all kinds to use, store, transmit, and display data.

e-business applications: XML implementations make electronic data interchange (EDI) more accessible for information interchange, business-to-business transactions, and business-to-consumer transactions.
Metadata applications: XML makes it easier to express metadata in a portable, reusable format.
Pervasive computing: XML provides portable and structured information types for display on pervasive (wireless) computing devices such as personal digital assistants (PDAs), cellular phones, and others. For example, WML (Wireless Markup Language) and VoiceXML are currently evolving standards for describing visual and speech-driven wireless device interfaces.

UNIT-IV

➢ Query Processing:

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

1. Parsing and translation
2. Optimization
3. Evaluation

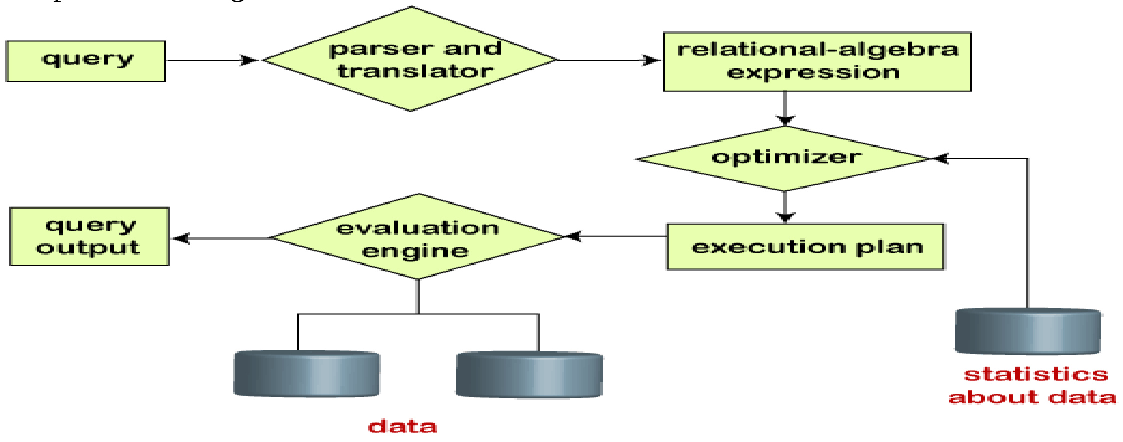
The query processing works in the following way:

Parsing and Translation

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

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

Exception Handling in



Steps in query processing

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

select emp_name from Employee where salary>10000;

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

- $\sigma_{salary>10000}$ (π_{salary} **(Employee))**
- π_{salary} ($\sigma_{salary>10000}$ **(Employee))**

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

➢ Estimating Query Cost/Measures of Query Cost

In the previous section, we understood about Query processing steps and evaluation plan. Though a system can create multiple plans for a query, the chosen method should be the best of all. It can be done by comparing each possible plan in terms of their estimated cost. For calculating the net estimated cost of any plan, the cost of each operation within a plan should be determined and combined to get the net estimated cost of the query evaluation plan.

The cost estimation of a query evaluation plan is calculated in terms of various resources that include:

- Number of disk accesses
- Execution time taken by the CPU to execute a query
- Communication costs in distributed or parallel database systems.

To estimate the cost of a query evaluation plan, we use the number of blocks transferred from the disk, and the number of disks seeks. Suppose the disk has an average block access time of t_s seconds and takes an average of t_T seconds to transfer x data blocks. The block access time is the sum of disk seeks time and rotational latency. It performs S seeks than the time taken will be $\mathbf{b*t_T + S*t_S}$ seconds. If t_T=0.1 ms, t_S =4 ms, the block size is 4 KB, and its transfer rate is 40 MB per second. With this, we can easily calculate the estimated cost of the given query evaluation plan.

Generally, for estimating the cost, we consider the worst case that could happen. The users assume that initially, the data is read from the disk only. But there must be a chance that the information is already present in the main memory. However, the users usually ignore this effect, and due to this, the actual cost of execution comes out less than the estimated value.

The response time, i.e., the time required to execute the plan, could be used for estimating the cost of the query evaluation plan. But due to the following reasons, it becomes difficult to calculate the response time without actually executing the query evaluation plan:

- When the query begins its execution, the response time becomes dependent on the contents stored in the buffer. But this information is difficult to retrieve when the query is in optimized mode, or it is not available also.
- When a system with multiple disks is present, the response time depends on an interrogation that in "what way accesses are distributed among the disks?". It is difficult to estimate without having detailed knowledge of the data layout present over the disk.
- Consequently, instead of minimizing the response time for any query evaluation plan, the optimizers finds it better to reduce the total **resource consumption** of the query plan. Thus to estimate the cost of a query evaluation plan, it is good to minimize the resources used for accessing the disk or use of the extra resources.

➢ Selection Operation in Query Processing

In the previous section, we understood that estimating the cost of a query plan should be done by measuring the total resource consumption.In this section, we will understand how the selection operation is performed in the query execution plan.

Generally, the selection operation is performed by the file scan. **File scans** are the search algorithms that for locating and accessing the data. It is the lowest-level operator used in query processing.

Let's see how selection using a file scan is performed.

Selection using File scans and Indices

In RDBMS or relational database systems, the file scan reads a relation only if the whole relation is stored in one file only. When the selection operation is performed on a relation whose tuples are stored in one file, it uses the following algorithms:

- **Linear Search:** In a linear search, the system scans each record to test whether satisfying the given selection condition. For accessing the first block of a file, it needs an initial seek. If the blocks in the file are not stored in contiguous order, then it needs some extra seeks. However, linear search is the slowest algorithm used for searching, but it is applicable in all types of cases. This algorithm does not care about the nature of selection, availability of indices, or the file sequence. But other algorithms are not applicable in all types of cases.

Selection Operation with Indexes

The index-based search algorithms are known as **Index scans**. Such index structures are known as **access paths**. These paths allow locating and accessing the data in the file. There are following algorithms that use the index in query processing:

- **Primary index, equality on a key:** We use the index to retrieve a single record that satisfies the equality condition for making the selection. The equality comparison is performed on the key attribute carrying a primary key.
- **Primary index, equality on nonkey:** The difference between equality on key and nonkey is that in this, we can fetch multiple records. We can fetch multiple records through a primary key when the selection criteria specify the equality comparison on a nonkey.
- **Secondary index, equality on key or nonkey:** The selection that specifies an equality condition can use the secondary index. Using secondary index strategy, we can either retrieve a single record when equality is on key or multiple records when the equality condition is on nonkey. When retrieving a single record, the time cost is equal to the primary index. In the case of multiple records, they may reside on different blocks. This results in one I/O operation per fetched record, and each I/O operation requires a seek and a block transfer.

Selection Operations with Comparisons

For making any selection on the basis of a comparison in a relation, we can proceed it either by using the linear search or via indices in the following ways:

- **Primary index, comparison:** When the selection condition given by the user is a comparison, then we use a primary ordered index, such as the primary B^+-tree index. **For example**, when A attribute of a relation R compared with a given value v as A>v, then we use a primary index on A to directly retrieve the tuples. The file scan starts its search from the beginning till the end and outputs all those tuples that satisfy the given selection condition.
- **Secondary index, comparison:** The secondary ordered index is used for satisfying the selection operation that involves <, >, ≤, or ≥ In this, the files scan searches the blocks of the lowest-level index. **(< ≤):** In this case, it scans from the smallest value up to the given value v. **(>, ≥):** In this case, it scans from the given value v up to the maximum value.

Implementing Complex Selection Operations

Working on more complex selection involves three selection predicates known as Conjunction, Disjunction, and Negation.

Conjunction: A conjunctive selection is the selection having the form as: $\boldsymbol{\sigma}_{\theta1\wedge\theta2\wedge\ldots\wedge\theta n}$ **(r)**

A conjunction is the intersection of all records that satisfies the above selection condition.

Disjunction: A disjunctive selection is the selection having the form as: $\boldsymbol{\sigma}_{\theta1\vee\theta2\vee\ldots\vee\theta n}$ **(r)**

A disjunction is the union of all records that satisfies the given selection condition $\boldsymbol{\theta_i}$.

Negation: The result of a selection $\boldsymbol{\sigma_{\neg\theta}(r)}$ is the set of tuples of given relation r where the selection condition evaluates to false. But nulls are not present, and this set is only the set of tuples in relation r that are not in $\boldsymbol{\sigma_\theta(r)}$.

Using these discussed selection predicates, we can implement the selection operations by using the following algorithms:

- **Conjunctive selection using one index:** In such type of selection operation implementation, we initially determine if any access path is available for an attribute. If found one, then algorithms based on the index will work better. Further completion of the selection operation is done by testing that each selected records satisfy the remaining simple conditions. The cost of the selected algorithm provides the cost of this algorithm.
- **Conjunctive selection via Composite index:** A composite index is the one that is provided on multiple attributes. Such an index may be present for some conjunctive selections. If the given selection operation proves true on the equality condition on two or more attributes and a composite index is present on these combined attribute fields, then directly search the index. Such type of index evaluates the suitable index algorithms.
- **Conjunctive selection via the intersection of identifiers:** This implementation involves record pointers or record identifiers. It uses indices with the record pointers on those fields which are involved in the individual selection condition. It scans each index for pointers to tuples satisfying the individual condition. Therefore, the intersection of all the retrieved pointers is the set of pointers to the tuples that satisfies the conjunctive condition. The algorithm uses these pointers to fetch the actual records. However, in absence of indices on each individual condition, it tests the retrieved records for the other remaining conditions.
- **Disjunctive selection by the union of identifiers:** This algorithm scans those entire indexes for pointers to tuples that satisfy the individual condition. But only if access paths are available on all disjunctive selection conditions. Therefore, the union of all fetched records provides pointers sets to all those tuples which satisfy or prove the disjunctive condition. Further, it makes use of pointers for fetching the actual records. Somehow, if the access path is not present for anyone condition, we need to use a linear search to find those tuples that satisfy the condition. Thus, it is good to use a linear search for determining such tests.

Cost Estimation Here, the overall cost of the algorithm is composed by adding the cost of individual index scans and cost of fetching the records in the intersection of the retrieved lists of pointers. We can fetch all selected records of the block using a single I/O operation because each pointer in the block appears together.

- The disk-arm movement gets minimized as blocks are read in sorted order.

➢ External Sort-Merge Algorithm

Till now, we saw that sorting is an important term in any database system. It means arranging the data either in ascending or descending order. We use sorting not only for generating a sequenced output but also for satisfying conditions of various database algorithms. In query processing, the sorting method is used for performing various relational operations such as joins, etc. efficiently. But the need is to provide a sorted input value to the system. For sorting any relation, we have to build an index on the sort key and use that index for reading the relation in sorted order. However, using an index, we sort the relation logically, not physically. Thus, sorting is performed for cases that include:

Case 1: Relations that are having either small or medium size than main memory.

Case 2: Relations having a size larger than the memory size.

In Case 1, the small or medium size relations do not exceed the size of the main memory. So, we can fit them in memory. So, we can use standard sorting methods such as quicksort, merge sort, etc., to do so.

For Case 2, the standard algorithms do not work properly. Thus, for such relations whose size exceeds the memory size, we use the External Sort-Merge algorithm.

The sorting of relations which do not fit in the memory because their size is larger than the memory size. Such type of sorting is known as **External Sorting**. As a result, the external-sort merge is the most suitable method used for external sorting.

External Sort-Merge Algorithm

Here, we will discuss the external-sort merge algorithm stages in detail:

In the algorithm, M signifies the number of disk blocks available in the main memory buffer for sorting.

Stage 1: Initially, we create a number of sorted runs. Sort each of them. These runs contain only a few records of the relation.

```
i = 0;
repeat
   read either M blocks or the rest of the relation having a smaller size;
   sort the in-memory part of the relation;
   write the sorted data to run file Ri;
   i =i+1;
Until the end of the relation
```

In Stage 1, we can see that we are performing the sorting operation on the disk blocks. After completing the steps of Stage 1, proceed to Stage 2.

Stage 2: In Stage 2, we merge the runs. Consider that total number of runs, i.e., N is less than M. So, we can allocate one block to each run and still have some space left to hold one block of output. We perform the operation as follows:

```
read one block of each of N files Ri into a buffer block in memory;
repeat
     select the first tuple among all buffer blocks (where selection is made in sorted order);
    write the tuple to the output, and then delete it from the buffer block;
                 if the buffer block of any run Ri is empty and not EOF(Ri)
   then read the next block of Ri into the buffer block;
Until all input buffer blocks are empty
```

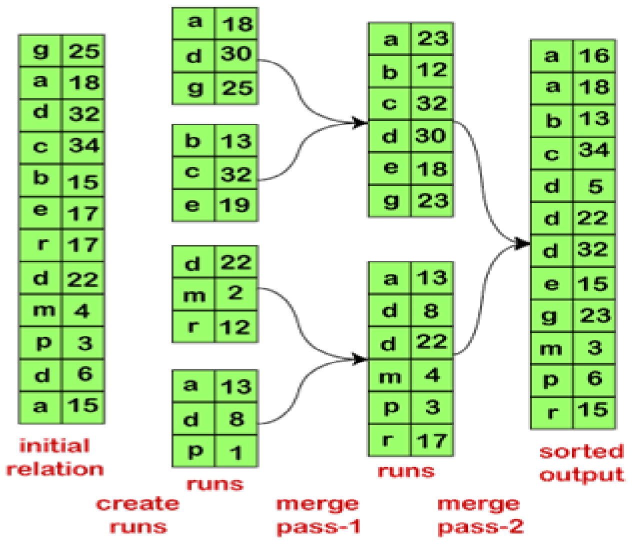
After completing Stage 2, we will get a sorted relation as an output. The output file is then buffered for minimizing the disk-write operations. As this algorithm merges N runs, that's why it is known as an **N-way merge.**

However, if the size of the relation is larger than the memory size, then either M or more runs will be generated in Stage 1. Also, it is not possible to allocate a single block for each run while processing Stage 2. In such a case, the merge operation process in multiple passes. As M-1 input buffer blocks have sufficient memory, each merge can easily hold M-1 runs as its input. So, the initial phase works in the following way:

- It merges the first M-1 runs for getting a single run for the next one.
- Similarly, it merges the next M-1 runs. This step continues until it processes all the initial runs. Here, the number of runs has a reduced M-1 value. Still, if this reduced value is greater than or equal to M, we need to create another pass. For this new pass, the input will be the runs created by the first pass.
- The work of each pass will be to reduce the number of runs by M-1 value. This job repeats as many times as needed until the number of runs is either less than or equal to M.
- Thus, a final pass produces the sorted output.

Example of External Merge-sort Algorithm

Let's understand the working of the external merge-sort algorithm and also analyze the cost of the external sorting with the help of an example.



External sorting using sort-merge

Suppose that for a relation R, we are performing the external sort-merge. In this, assume that only one block can hold one tuple, and the memory can hold at most three blocks. So, while processing Stage 2, i.e., the merge stage, it will use two blocks as input and one block for output.

Joins

Several different algorithms to implement joins

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

Nested-Loop Join

To compute the theta join r θ s

```
for each tuple t r in r do begin
for each tuple t s in s do begin
test pair (t r ,ts) to see if they satisfy the join condition θ
if they do, add t r • ts to the result
end
end
```

r is called the outer relation and s the inner relation of the join.
Requires no indices and can be used with any kind of join condition.
Expensive since it examines every pair of tuples in the two relations.
In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is nr * bs + br block transfers, plus
nr + br seeks
If the smaller relation fits entirely in memory, use that as the inner relation.
Reduces cost to br + bs block transfers and 2 seeks
Assuming worst case memory availability cost estimate is
with student as outer relation:
5000 * 400 + 100 = 2,000,100 block transfers,
5000 + 100 = 5100 seeks
with takes as the outer relation
10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
If smaller relation (student) fits entirely in memory, the cost estimate will be 500 block transfers.

Block Nested-Loop Join

Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block Br of r do begin
    for each block Bs of s do begin
      for each tuple t r in Br do begin
          for each tuple t s in Bs do begin
            Check if (t r ,ts) satisfy the join condition
               End
          End
    End
  End
```

Worst case estimate: br * bs + br block transfers + 2 * br seeks
Each block in the inner relation s is read once for each block in the outer relation
Best case: br + bs block transfers + 2 seeks.
Improvements to nested loop and block nested loop algorithms:
In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
Cost = $\lceil br / (M-2) \rceil$ * bs + br block transfers + 2 $\lceil br / (M-2) \rceil$ seeks
If equi-join attribute forms a key or inner relation, stop inner loop on first match
Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)

Indexed Nested-Loop Join

Index lookups can replace file scans if join is an equi-join or natural join and an index is available on the inner relation's join attribute .

Can construct an index just to compute a join.
For each tuple t r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t r .
Worst case: buffer has space for only one page of r, and, for each tuple in r, we perform an index lookup on s. Cost of the join: br (tT + tS) + nr * c Where c is the cost of traversing index and fetching all matching s tuples for one tuple or r. c can be estimated as cost of a single selection on s using the join condition.
If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Merge-Join

Sort both relations on their join attribute (if not already sorted on the join attributes)
Merge the sorted relations to join them
Join step is similar to the merge stage of the sort-merge algorithm.
Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
Can be used only for equi-joins and natural joins
Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory.
hybrid merge-join: If one relation is sorted, and the other has a secondary B+-tree index on the join attribute.
Merge the sorted relation with the leaf entries of the B+-tree.
Sort the result on the addresses of the unsorted relation's tuples
Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples. Sequential scan more efficient than random lookup

Hash-Join

Hash Join Algorithm
The Hash Join algorithm is used to perform the natural join or equi join operations. The concept behind the Hash join algorithm is to partition the tuples of each given relation into sets. The partition is done on the basis of the same hash value on the join attributes. The hash function provides the hash value. The main goal of using the hash function in the algorithm is to reduce the number of comparisons and increase the efficiency to complete the join operation on the relations.
For example, suppose there are two tuples a and b where both of them satisfy the join condition. It means they have the same value for the join attributes. Suppose that both a and b tuples consist of a hash value as i. It implies that tuple a should be in a_i, and tuple b should be in b_i. Thus, we only compare a tuples in a_i with b tuples of b_i. We do not need to compare the b tuples in any other partition. Therefore, in this way, the hash join operation works.
Hash Join Algorithm

```
//Partition s//
for each tuple $t_s$ in s do begin
i = h($t_s$ [JoinAttrs]);
Hsi = Hsi U {$t_s$};
end
//Partition r//
for each tuple $t_r$ in r do begin
i = h($t_r$[JoinAttrs]);
Hri = Hri U {$t_r$};
```

```
end
//Perform the join operation on each partition//
for i= 0 to nh do begin
        read Hsi and build an in-memory hash index on it;
        for each tuple $t_r$ in Hri do begin
          probe the hash index on Hsi to locate all tuples
              such that $t_s$[JoinAttrs] = $t_r$[JoinAttrs];
          for each matching tuple $t_s$ in Hsi do begin
                add $t_r \bowtie t_s$ to the result;
          end
        end
end
```

It is the Hash join algorithm in which we have computed the natural join of two given relations r and s. In the algorithm, there are various terms used:

$t_r \bowtie t_s$: It defines the concatenation of the attributes of tuple t_r and t_s, which is further followed by projecting out the repeated attributes.

t_r and t_s: These are the tuples of relations r and s, respectively.

➢ Evaluation of Expressions

In our previous sections, we understood various concepts in query processing. We learned about the processing steps, selection operations, and also several types of algorithms used for performing th operation with their cost estimations.

We are already aware of computing and representing the individual relational operations for the given user or expression. Here, we will get to know how to compute and evaluate an expression with multiple operatic For evaluating an expression that carries multiple operations in it, we can perform the computation o operation one by one. However, in the query processing system, we use two methods for evaluati expression carrying multiple operations. These methods are:

1. Materialization
2. Pipelining

Let's take a brief discussion of these methods.

Materialization

In this method, the given expression evaluates one relational operation at a time. Also, each operation is ev in an appropriate sequence or order. After evaluating all the operations, the outputs are materialize temporary relation for their subsequent uses. It leads the materialization method to a disadvantag disadvantage is that it needs to construct those temporary relations for materializing the results of the ev operations, respectively. These temporary relations are written on the disks unless they are small in size.

Pipelining

Pipelining is an alternate method or approach to the materialization method. In pipelining, it enables evaluate each relational operation of the expression simultaneously in a pipeline. In this approach, after eva one operation, its output is passed on to the next operation, and the chain continues till all the relational ope are evaluated thoroughly. Thus, there is no requirement of storing a temporary relation in pipelining. S advantage of pipelining makes it a better approach as compared to the approach used in the material method. Even the costs of both approaches can have subsequent differences in-between. But, both appı perform the best role in different cases. Thus, both ways are feasible at their place.

➢ Query Optimization

A query optimizer translates a query into a sequence of physical operators that can be directly carried ou the query execution engine. The output of the optimizer is called a query execution plan. The execution may be thought of as a dataflow datagram that pipes data through a graph of query operators. The goal of q optimization is to derive an efficient execution plan in terms of relevant performance measures, suc memory usage and query response time. To achieve this, the optimizer needs to provide: (i) a spac execution plans (search space), (ii) cost estimation techniques to assign a relevant cost to each plan in search space, and (iii) an enumeration algorithm to search through the space of plans.

The query optimizer takes as input a parsed query and produces as output an efficient execution plan fo query. The task of the optimizer is nontrivial, since given a query (i) there are many logically equivalent...

Optimization

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

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

➢ Transforming Relational Expressions

The first step of the optimizer says to implement such expressions that are logically equivalent to the expression. For implementing such a step, we use the equivalence rule that describes the method to transfo generated expression into a logically equivalent expression.

Although there are different ways through which we can express a query, with different costs. But for expr a query in an efficient manner, we will learn to create alternative as well as equivalent expressions of the expression, instead of working with the given expression. Two relational-algebra expressions are equiva both the expressions produce the same set of tuples on each legal database instance. A **legal dat instance** refers to that database system which satisfies all the integrity constraints specified in the da schema. However, the sequence of the generated tuples may vary in both expressions, but they are cons equivalent until they produce the same tuples set.

Equivalence Rules

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

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

θ, θ_1, θ_2 … : Used for denoting the predicates.

L_1, L_2, L_3 … : Used for denoting the list of attributes.

E, E_1, E_2 …. : Represents the relational-algebra expressions.

Let's discuss a number of equivalence rules:

Rule 1: Cascade of σ

This rule states the deconstruction of the conjunctive selection operations into a sequence of individual selections. Such a transformation is known as a **cascade of σ**.

$\sigma_{\theta 1 \wedge \theta 2} (E) = \sigma_{\theta 1} (\sigma_{\theta 2} (E))$
Rule 2: Commutative Rule
a) This rule states that selections operations are commutative.
$\sigma_{\theta 1} (\sigma_{\theta 2} (E)) = \sigma_{\theta 2} (\sigma_{\theta 1} (E))$
b) Theta Join (θ) is commutative.
$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$ (θ is in subscript with the join symbol)
However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.
However, in the case of theta join, the equivalence rule does not work if the order of attributes is considered. Natural join is a special case of Theta join, and natural join is also commutative.
Rule 3: Cascade of ∏
This rule states that we only need the final operations in the sequence of the projection operations, and other operations are omitted. Such a transformation is referred to as a **cascade of ∏**.
∏L1 (∏L2 (. . . (∏Ln (E)) . . .)) = ∏L1 (E)
Rule 4: We can combine the selections with Cartesian products as well as theta joins
Rule 4: We can combine the selections with Cartesian products as well as theta joins
1. ***$\sigma_{\theta} (E_1 \times E_2) = E_{1\theta} \bowtie E_2$***
2. ***$\sigma_{\theta 1} (E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$***

Rule 5: Associative Rule
a) This rule states that natural join operations are associative.***$(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$***
b) Theta joins are associative for the following expression:
$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$
In the theta associativity, θ2 involves the attributes from E2 and E3 only. There may be chances
Rule 6: Distribution of the Selection operation over the Theta join.
Under two following conditions, the selection operation gets distributed over the theta-join operation:
a) When all attributes in the selection condition θ_0 include only attributes of one of the expressions which are being joined.
$\sigma_{\theta 0} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0} (E_1)) \bowtie_{\theta} E_2$
b) When the selection condition θ_1 involves the attributes of E_1 only, and θ_2 includes the attributes of E_2 only.***$\sigma_{\theta 1 \wedge \theta 2} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1} (E_1)) \bowtie_{\theta} ((\sigma_{\theta 2} (E_2))$***
Rule 7: Distribution of the projection operation over the theta join.
Under two following conditions, the selection operation gets distributed over the theta-join operation:
a) Assume that the join condition θ includes only in L_1 υ L_2 attributes of E_1 and E_2 Then, we get the following expression:***$\prod_{L1 \upsilon L2} (E_1 \bowtie_{\theta} E_2) = (\prod_{L1} (E_1)) \bowtie_{\theta} (\prod_{L2} (E_2))$***
b) Assume a join as $E_1 \bowtie E_2$. Both expressions E_1 and E_2 have sets of attributes as L_1 and L_2. Assume two attributes L_3 and L_4 where L_3 be attributes of the expression E_1, involved in the θ join condition but not in L_1 υ L_2 Similarly, an L_4 be attributes of the expression E_2 involved only in the θ join condition and not in L_1 υ L_2 attributes. Thus, we get the following expression:
$\prod_{L1 \upsilon L2} (E_1 \bowtie_{\theta} E_2) = \prod_{L1 \upsilon L2} ((\prod_{L1 \upsilon L3} (E_1)) \bowtie_{\theta} ((\prod_{L2 \upsilon L4} (E_2)))$
Rule 8: The union and intersection set operations are commutative.
$E_1 \upsilon E_2 = E_2 \upsilon E_1$
$E_1 \cap E_2 = E_2 \cap E_1$
However, set difference operations are not commutative.
Rule 9: The union and intersection set operations are associative.
$(E_1 \upsilon E_2) \upsilon E_3 = E_1 \upsilon (E_2 \upsilon E_3)$

$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

Rule 10: Distribution of selection operation on the intersection, union, and set difference operations. The below expression shows the distribution performed over the set difference operation.

$\sigma_p (E_1 - E_2) = \sigma_p(E_1) - \sigma_p(E_2)$

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

$\sigma_p (E_1 - E_2) = \sigma_p(E_1) - E_2$

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

This rule states that we can distribute the projection operation on the union operation for t expressions. $\prod_L (E_1 \cup E_2) = (\prod_L (E_1)) \cup (\prod_L (E_2))$

➢ Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as *a* (*b c*) to estimate the cost of joining *a* with (*b c*), we need to have estimates of statistics such as the size of *b c*.

A query evaluation plan that has the lowest estimated execution cost may therefore not actually have the lowest actual execution cost. However, real-world experience has shown that even if estimates are not precise, the plans with the lowest estimated costs usually have actual execution costs that are either the lowest actual execution costs, or are close to the lowest actual execution costs.

Catalog Information

The DBMS catalog stores the following statistical information about database relations:

• nr , the number of tuples in the relation r.

• br , the number of blocks containing tuples of relation r.

• lr , the size of a tuple of relation r in bytes.

• fr , the blocking factor of relation *r* — that is, the number of tuples of relation *r* that fit into one block.

V (A, r), the number of distinct values that appear in the relation r for attribute A. This value is the same as the size of ΠA(r). If A is a key for relation r, V (A, r) is nr .

The last statistic, *V* (*A*, *r*), can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, *A*, *V* (*A*, *r*) is the size of Π*A*(*r*).

Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. For instance, some databases store the distribution of values for each attribute as a **histogram**: in a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. As an example of a histogram, the range of values for an attribute *age* of a relation *person* could be divided into 0 – 9, 10 – 19, …, 90 – 99 (assuming a maximum age of 99). With each range we store a count of the number of *person* tuples whose *age* values lie in that range. With- out such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same count.

Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

• *σA* = *a*(*r*): If we assume uniform distribution of values (that is, each value appears with equal probability), the selection result can be estimated to have *nr* /*V* (*A*, *r*) tuples, assuming that the value *a* appears in attribute *A* of some record of *r*. The assumption that the value *a* in the selection appears in some record is generally true, and cost estimates often make it implicitly.

• $\sigma_{A \leq v}(r)$: Consider a selection of the form $\sigma_{A \leq v}(r)$. If the actual value used in the comparison (v) is available at the time of cost estimation, a more ac- curate estimate can be made. The lowest and highest values (min(A, r) and max(A, r)) for the attribute can be stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will satisfy the condition $A \leq v$ as 0 if $v <$ min(A, r), as nr if $v \geq$ max(A, r), and otherwise.

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

Complex selections:

Conjunction: A conjunctive selection is a selection of the form

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

We can estimate the result size of such a selection: For each θi, we estimate the size of the selection $\sigma\theta i$ (r), denoted by si, as described previously. Thus, the probability that a tuple in the relation satisfies selection condition θi is si/nr .

Negation: In the absence of nulls, the result of a selection $\sigma\neg\theta$ (r) is simply the tuples of r that are not in $\sigma\theta$ (r). We already know how to estimate the number of tuples in $\sigma\theta$ (r). The number of tuples in $\sigma\neg\theta$ (r) is therefore estimated to be n(r) minus the estimated number of tuples in $\sigma\theta$ (r).

We can account for nulls by estimating the number of tuples for which the condition θ would evaluate to *unknown*, and subtracting that number from the above estimate ignoring nulls. Estimating that number would require extra statistics to be maintained in the catalog.

Join Size Estimation the Cartesian product $r \times s$ contains $nr * ns$ tuples. Each tuple of $r \times s$ occupies $lr + ls$ bytes, from which we can calculate the size of the Cartesian product.

Estimating the size of a natural join is somewhat more complicated than estimating the size of a selection or of a Cartesian product. Let $r(R)$ and $s(S)$ be relations.

If $R \cap S = \emptyset$ — that is, the relations have no attribute in common — then r s is the same as $r \times s$, and we can use our estimation technique for Cartesian products.

• If $R \cap S$ is a key for R, then we know that a tuple of s will join with at most one tuple from r. Therefore, the number of tuples in rs I s no greater than the number of tuples in s. The case where $R \cap S$ is a key for S is symmetric to the case just described. If $R \cap S$ forms a foreign key of S, referencing R, the number of tuples in r s is exactly the same as the number of tuples in s.

• The most difficult case is when $R \cap S$ is a key for neither R nor S. In this case, we assume, as we did for selections, that each value appears with equal probability.

Size Estimation for Other Operations

We outline below how to estimate the sizes of the results of other relational algebra operations.

Projection: The estimated size (number of records or number of tuples) of a projection of the form $\Pi A(r)$ is $V(A, r)$, since projection eliminates duplicates. **Aggregation:** The size of AGF (r) is simply $V(A, r)$, since there is one tuple in AGF (r) for each distinct value of A.

Set operations: If the two inputs to a set operation are selections on the same relation, we can rewrite the set operation as disjunctions, conjunctions, or negations. For example, $\sigma\theta 1$ (r) $\cup \sigma\theta 2$ (r) can be rewritten as $\sigma\theta 1$ $\vee\theta 2$ (r). Similarly, we can rewrite intersections as conjunctions, and we can rewrite set difference by using negation, so long as the two relations participating in the set operations are selections on the same relation. We can then use the estimates for selections involving conjunctions, disjunctions, and negation in Section 14.2.2.

Estimation of Number of Distinct Values

The estimates of distinct values are straightforward for projections: They are the same in $\Pi A(r)$ as in r. same holds for grouping attributes of aggregation. For results of **sum**, **count**, and **average**, we can assume simplicity, that all aggregate values are distinct. For **min**(A) and **max**(A), the number of distinct values ca

estimated as min($V(A, r), V(G, r)$), where G denotes the grouping attributes. We omit details of estima
distinct values for other operations.

➢ Choice of Evaluation Plan

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

How they are evaluated using different methods and what the different costs are when different methods are used. Now the important phase while evaluating a query is deciding which evaluation plan has to be selected so that it can be traversed efficiently. It collects all the statistics, costs, access/ evaluation paths, relational trees etc. It then analyses them and chooses the best evaluation path.

Same query is written in different forms of relational algebra. Corresponding trees for them too is drawn by DBMS. Statistics for them based on cost based evaluation and heuristic methods are collected. It checks the costs based on the different techniques that we have seen so far. It checks for the operator, joining type, indexes, number of records, selectivity of records, distinct values etc from the data dictionary. Once all these informations are collected, it picks the best evaluation plan.

Have look at below relational algebra and tree for EMP and DEPT.

∏ EMP_ID, DEPT_NAME (σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' (EMP) ∞DEPT)
Or
∏ EMP_ID, DEPT_NAME (σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' (EMP ∞DEPT))
Or
σ DEPT_ID = 10 AND EMP_LAST_NAME = 'Joseph' (∏ EMP_ID, DEPT_NAME, DEPT_ID (EMP ∞DEPT))

What can be observed here: First tree reduces the number of records for joining and seems to be efficient. But what happens if we have index on DEPT_ID? Then the join between EMP and EMP can also be more efficient. But we see the filter condition on EMP table, we have DEPT_ID = 10, which is index column. Hence first applying selection condition and then join will reduce the number of records as well as make the join more efficient than without index. Next are the projected columns – EMP_ID and DEPT_NAME. they are all distinct values. There cannot be duplicate values for them. But we are selecting those values for DEPT_ID = 10, hence DEPT_NAME has only one value. Hence their selectivity is same as number of employees working for DEPT_ID = 10. But we are selecting only those employees whose last name is 'Joseph'. Hence the selectivity is min (distinct (employee (DEPT_10)), distinct (employee (DEPT_10, JOSEPH)). Obliviously distinct (employee (DEPT_10, JOSEPH)) would have lesser value. The optimizer decides all these factors for above 3 trees and then decides first tree would be more efficient. Hence it evaluates the query using first tree