III YEAR – V SEMESTER

COURSE CODE: 7BCE5C2

CORE COURSE-X–RELATIONAL DATABASE MANGEMENT SYSTEMS

Unit I

Introduction: Database System Applications – Purpose of Database Systems – View of Data– Database Languages – Relational Databases – Database Design – Object based and semi structured databases – Data storage and Querying – Database Users and Administrators– Transaction Management – Database users and Architectures – History of Database System.

Entity-Relationship Model: E-R model – constraints – E-R diagrams – E-R design issues – weak entity sets – Extended E-R features.

Unit II

Relational Database Design: Features of good Relational designs – Atomic domains and First Normal Form – Decomposition using functional dependencies – Functional dependency theory – Decomposition using functional – Decomposition using multivalued dependencies – more Normal forms – database design process – modeling temporal data

Unit III

Database System Architecture: Centralized and Client-Server architecture – Server system architecture – parallel systems – Distributed systems – Network types. Parallel databases: I/O parallelism – Interquery Parallelism – Intraquery parallelism. Distributed Databases: Homogeneous and Heterogeneous databases – Distributed Data storage – Distributed transactions – Distributed query processing.

Unit IV

Schema Objects Data Integrity – Creating and Maintaining Tables – Indexes – Sequences – Views – Users Privileges and Roles –Synonyms.

Unit V

PL/SQL: PL/SQL – Triggers – Stored Procedures and Functions – Package – Cursors – Transaction

Text Books:

1. Database System Concepts – SilberschatzKorthSudarshan, International (5th Edition) McGraw Hill Higher Education 2006
2. Jose A.Ramalho – Learn ORACLE 8i BPB Publications 2003

Books for Reference:

1. "Oracle 9i The complete reference", Kevin Loney and George Koch, Tata McGraw Hill, 2004.
2. "Database Management Systems", Ramakrishnan and Gehrke, Mc Graw Hill, Third Edition, 2003.
3. "Oracle 9i PL/SQL Programming "Scott Urman, Oracle Press, Tata Mc Graw Hill, 2002.

♣♣♣♣♣♣♣♣

UNIT I

Introduction

- Database System Applications
- Purpose of Database Systems
- View of Data
- Database Languages

- Relational Databases
- Database Design
- Object based and semi structured databases
- Data storage and Querying
- Database Users and Administrators
- Transaction Management
- Database users and Architectures
- History of Database System.

Entity-Relationship Model:

- E-R model
- Constraints
- E-R diagrams
- E-R design issues
- Weak entity sets
- Extended E-R features.

Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

What is a Database?

To find out what database is, we have to start from data, which is the basic building block of any DBMS.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).

Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	Name	Age
1	ABC	19

Table or **Relation**: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called **Fields**, **Attributes** or **Domains**. The rows are called **Tuples**
or **Records**.

Database: Collection of related relations. Consider the following collection of tables:

T1

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3

Roll	Year
1	I
2	II
3	I

T4

Year	Hostel
I	H1
II	H2

We now have a collection of 4 tables. They can be called a "related collection" because we can clearly find out that there are some common attributes existing in a selected pair of tables. Because of these common attributes we may combine the data of two or more tables together to find out the complete details of a student. Questions like "Which hostel does the youngest student live in?" can be answered now, although

Age and ***Hostel*** attributes are in different tables.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

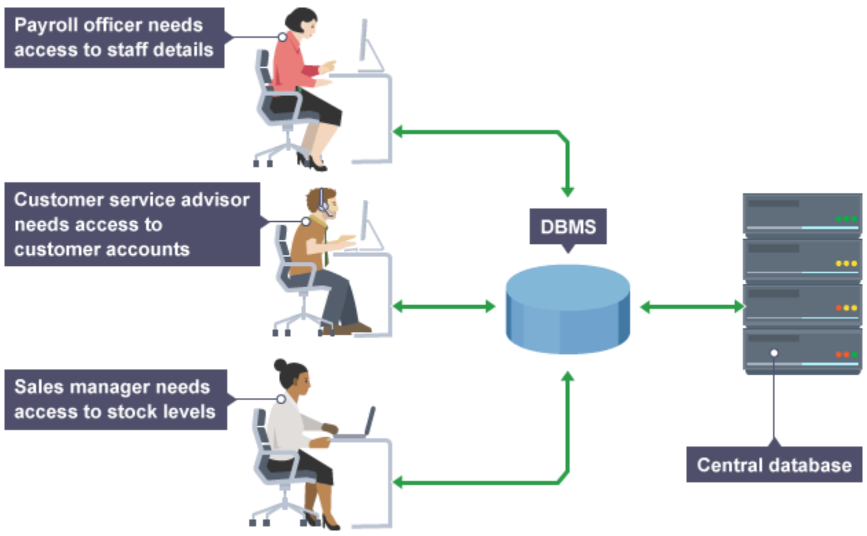

Payroll officer needs access to staff details
Customer service advisor needs access to customer accounts
Sales manager needs access to stock levels
DBMS
Central database

Figure 1.1: Empolyees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

In a database, data is organized strictly in row and column format. The rows are called **Tuple** or **Record**. The data items within one row may belong to different data types. On the other hand, the columns are often called **Domain** or **Attribute**. All the data items within a single attribute are of the same data type.

What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data,** we mean known facts that can be recorded and that have implicit meaning.

Database Management System (DBMS) and Its Applications:

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

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. 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 generation of paychecks.

◦ *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items inwarehouses and stores, and orders for items.

Online retailers: For sales data noted above plus online order tracking,generation of recommendation lists, 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 stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- *Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on 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 commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- ✓ Addnew students, instructors, and courses
- ✓ Register students for courses and generate class rosters
- ✓ Assign grades to students, compute grade point averages (GPA), and generate transcripts

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

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

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

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

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers.

As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to 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 example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to 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, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

Advantages of DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

Improved Data Sharing : DBMS allows a user to share the data in any number of application programs.

Data Integrity : Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can can enforce an integrity that it must accept the customer only from Noida and Meerut city.

Security : Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

Data Consistency : By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

Efficient Data Access : In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

Enforcements of Standards : With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

Data Independence : Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

Reduced Application Development and Maintenance Time : DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out ofit.

2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.

3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.

4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the 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 designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

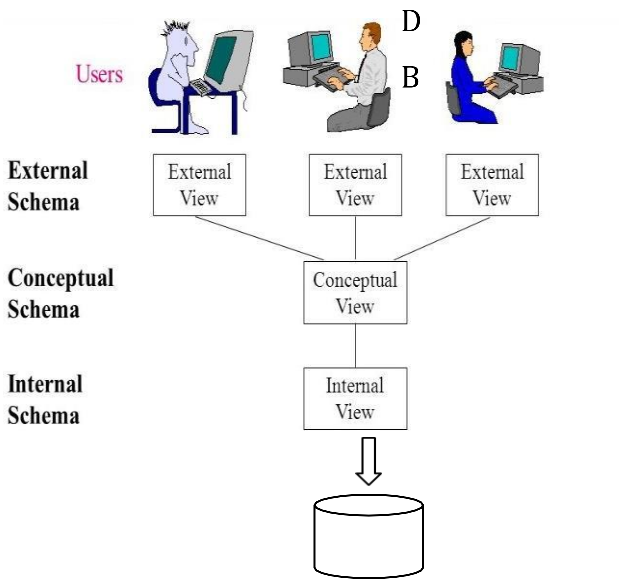


Figure 1.2 : Levels of Abstraction in a DBMS

• **Physical level (or Internal View / Schema)**: The lowest level of abstraction describes *how* the data are actually 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 are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database 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 entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database. Figure 1.2 shows the relationship among the three levels of abstraction.

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

```
      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 type 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*, and *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 consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the 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 code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

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

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database.

The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all. The concept of database schemas and instances can be understood by analogy to a program written in a programming language.

A database schema corresponds to the variable declarations (along with associated type definitions) in a program.

Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitioned according to the levels of abstraction.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on thephysical schema, and thus need not be rewritten if the physical schema changes.

Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

Relational Model. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.
Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Entity-Relationship Model. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.
An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity- relationship model is widely used in database design.

Object-Based Data Model. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

Semi-structured Data Model. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semi-structured data.

Historically, the **network data model** and the **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places.

Database Languages

A database system provides a **data-definition language** to specify the database schema and a **data-manipulation language** to express database queries and updates. In practice, the data- definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data-Manipulation Language

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion ofinformation from the database
- Modification of information stored in the database

There are basically two types:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

Data-Definition Language (DDL)

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**). The DDL is also used to specify additional properties of the data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain **consistency constraints**.
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimal overhead.

• **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

• **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation.
Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

• **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

• **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata—**that is, data about data. The data dictionary is considered to be a special type of table that can only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.

Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date. A passive data dictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

Database Administrators and DatabaseUsers

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data. **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Database Architecture:

We are now in a position to provide a single picture (Figure 1.3) of the various components of a database system and the connections among them.

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

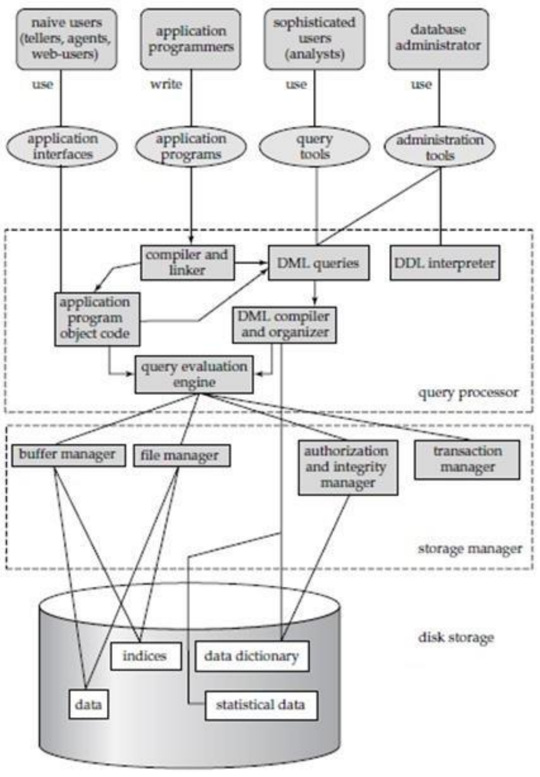


Figure 1.3: Database System Architecture

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components. The storage manager is important because databases typically require a large amount of storage space. The query processor is important because it helps the database system simplify and facilitate access to data.

It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Database applications are usually partitioned into two or three parts, as in Figure 1.4. In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface.

The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the WorldWideWeb.

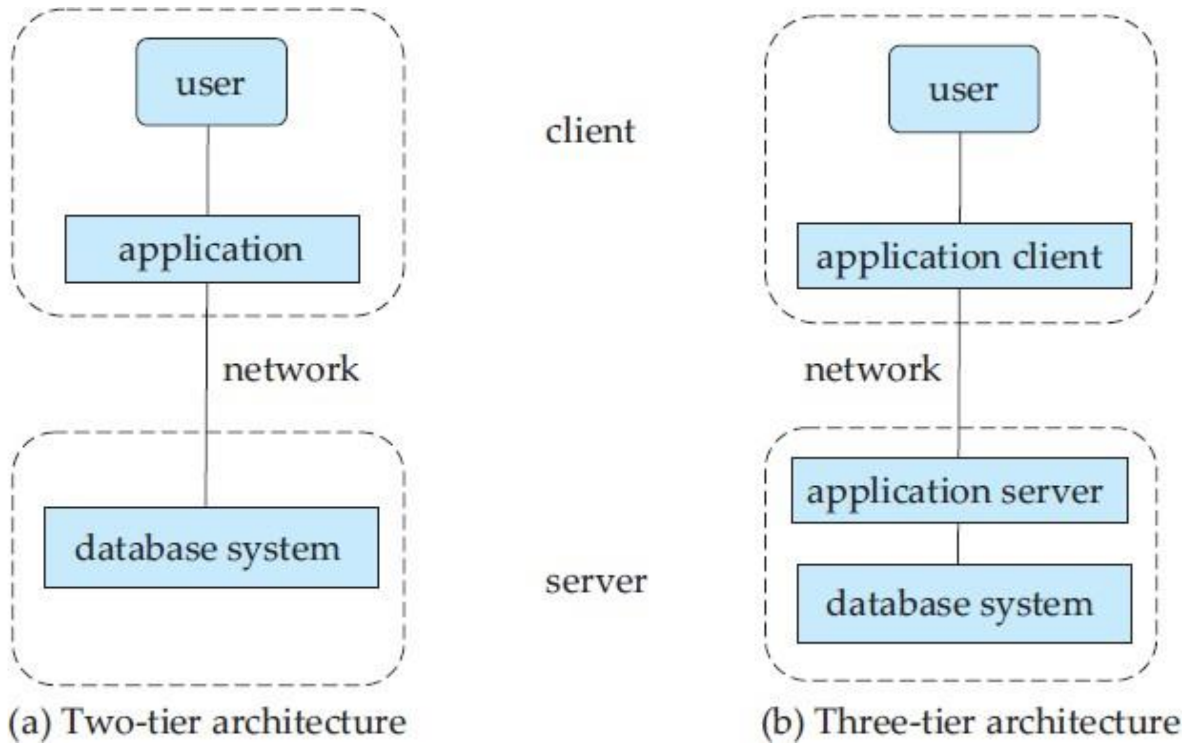


Figure 1.4: Two-tier and three-tier architectures.

Query Processor:

The query processor components include

· **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.

· **DML compiler,** which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

Storage Manager:

A *storage manager* is a program module that provides the interface between the lowlevel data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

· **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

· **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

· **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

· **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

Transaction Manager:

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. **Transaction - manager** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

Entity-Relationship Model

- Design Process
- Modeling
- Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML

Modeling

A *database* can be modeled as: a collection of entities, relationship among entities.

An **entity** is an object that exists and is distinguishable fromother objects.

Example: specific person, company, event, plant

Entities have *attributes,* Example: people have *names* and *addresses*

An **entity set** is a set of entities of the same type that share the same properties.

Example: set of all persons, companies, trees, holidays

Relationship Sets

A **relationship** is an association among several entities Example:

 A **relationship set** is a mathematical relation among n 2 entities, each taken from entity sets

$\{(e_1, e_2, \ldots e_n) \mid e_1$ E_1, e_2 $E_2, \ldots, e_n$ $E_n\}$

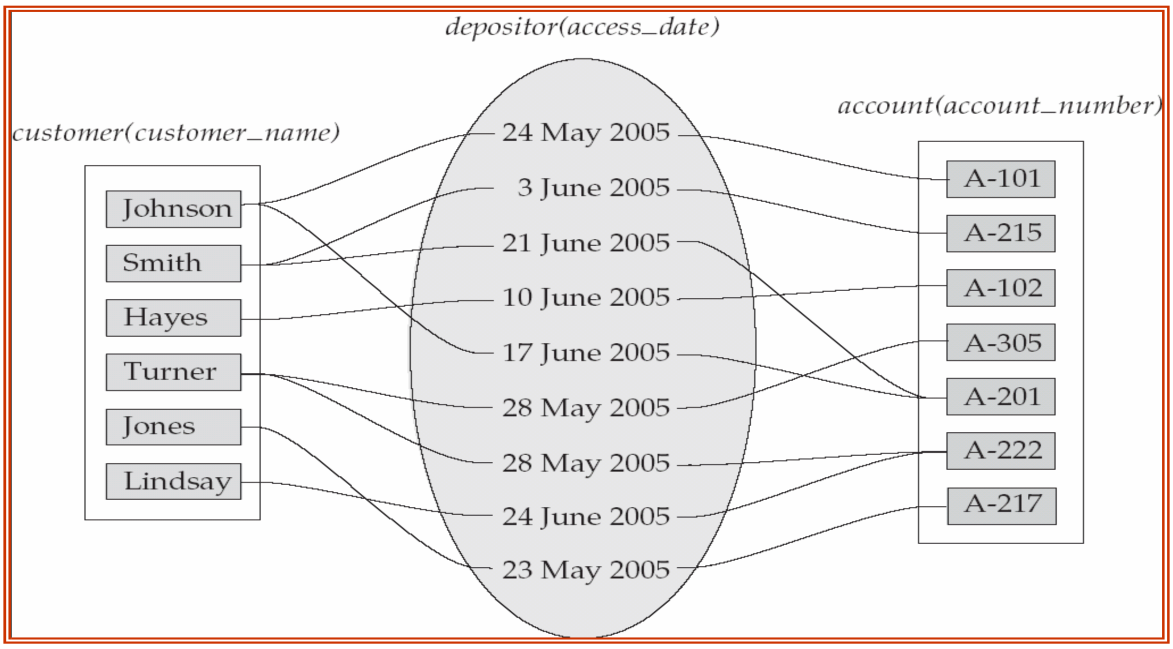
where $(e_1, e_2, \ldots, e_n)$ is a relationship

 Example:

(Hayes, A-102) *depositor*

 An **attribute** can also be property of a relationship set.

 For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date*



Degree of a Relationship Set

- Refers to number of entity sets that participate in arelationship set.
- Relationship sets that involve two entity sets are **binary** (or degree two). Generally, most relationship sets in a database system are binary.
- Relationship sets may involve more than two entity sets.

  Example: Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee, job, and branch*

Relationships between more than two entity sets are rare. Most relationships are binary.

Attributes

- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

  Example:

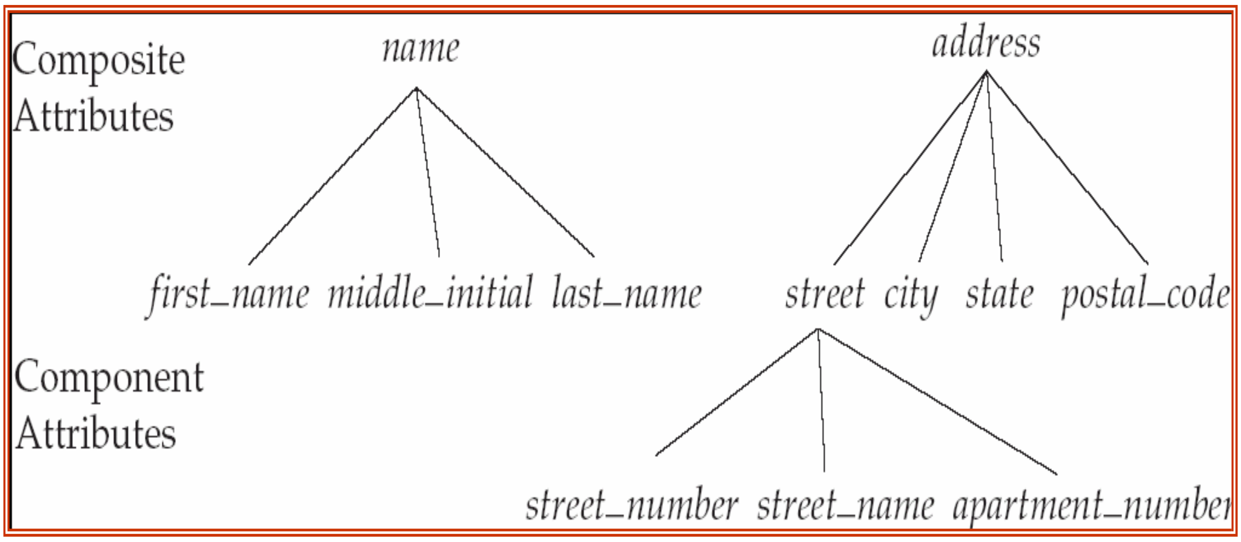
  customer = (*customer_id, customer_name,* customer_street, customer_city)

  loan = (*loan_number, amount*)
- **Domain** – the set of permitted values for each attribute
- Attribute types:
  - *Simple* and *composite* attributes.
  - *Single-valued* and *multi-valued* attributes
    - Example: multivalued attribute: *phone_numbers*
  - *Derived* attributes

Can be computed from other attributes

Example: age, given date_of_birth

Composite Attributes



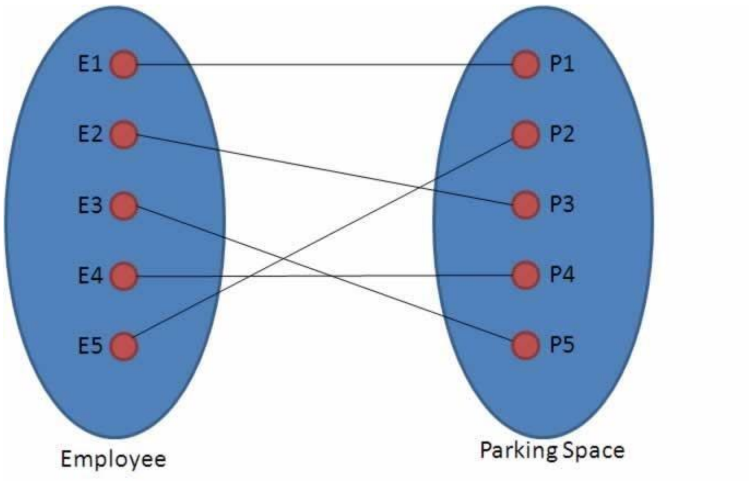
Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one

- Many to many

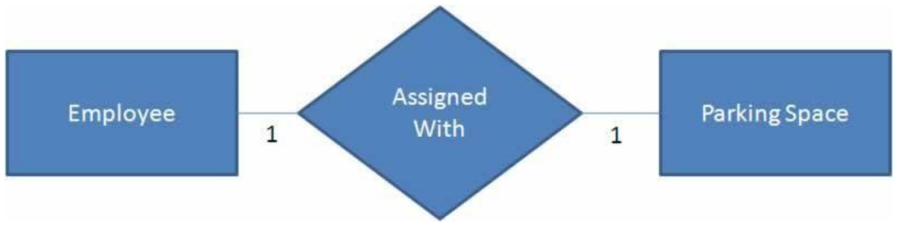
Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.



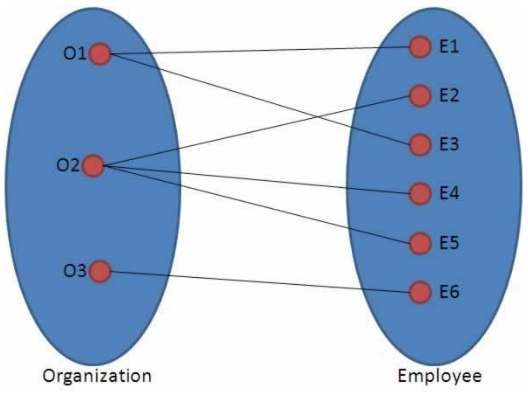
One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below



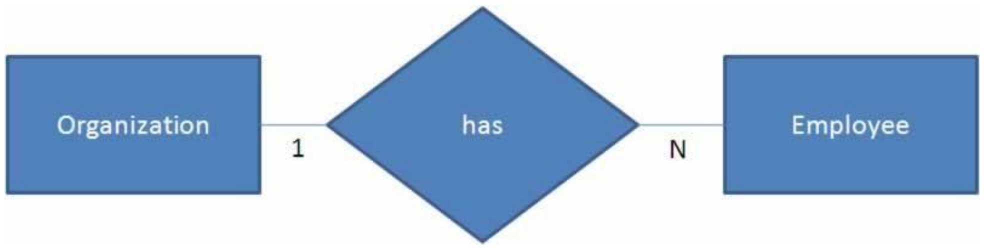
Example for Cardinality – One-to-Many (1:N)

Organization has employees



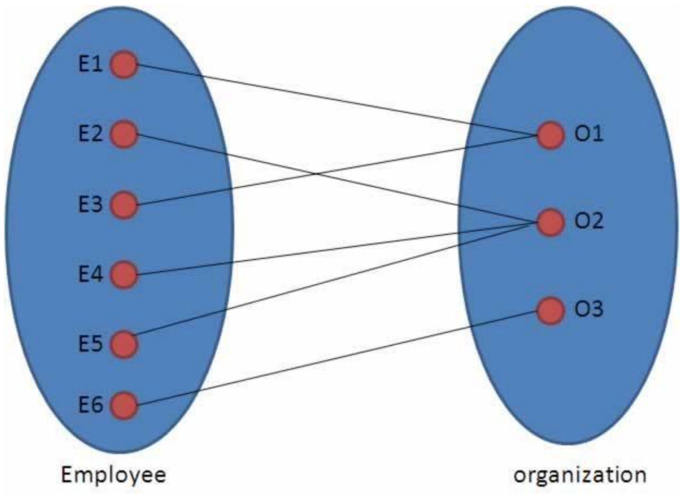
One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below



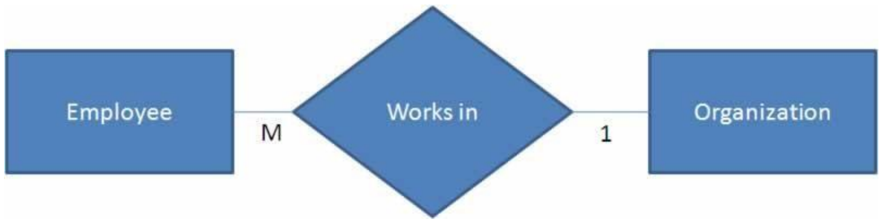
Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization



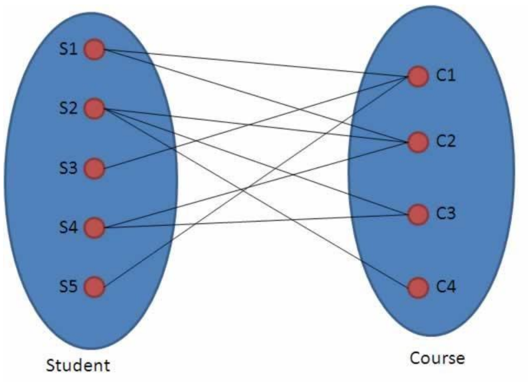
One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.



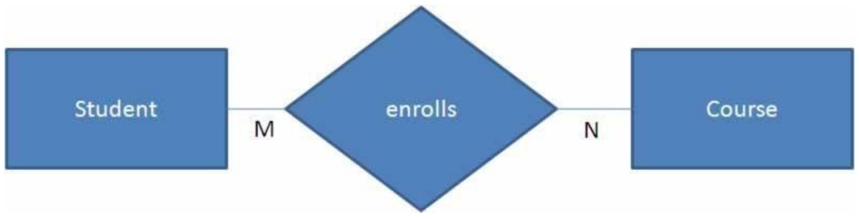
Cardinality – Many-to-Many (M:N)

Students enrolls for courses



One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

In ER modeling, this can be mentioned using notations as given below



Design Issues

Use of entity sets vs. attributes

Choice mainly depends on the structure of the enterprisebeing modeled, and on the semantics associated with the attribute in question.

Use of entity sets vs. relationship sets

Possible guideline is to designate a relationship set to describean action that occurs between entities

Binary versus n-ary relationship sets

Although it is possible to replace any nonbinary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets,a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.

Weak Entity Sets

1. An entity set that does not have a primary key is referred to as a **weak entity set**.
2. The existence of a weak entity set depends on the existence of a identifying entity set

   I) it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

   II) Identifying relationship depicted using a double diamond
3. The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entityset.
4. The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.
5. We depict a weak entity set by double rectangles.
6. We underline the discriminator of a weak entity set with a dashed line.
7. payment_number – discriminator of the *payment* entity set
8. Primary key for *payment* – (*loan_number, payment_number*)

Extended E-R Features: Specialization

Top-down design process; we designate subgroupings within an entityset that are distinctive from other entities in the set.

These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entityset.

Depicted by a *triangle* component labeled ISA (E.g. *customer* "is a"*person*).

> **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Generalization

A bottom-up design process – combine a number of entity sets that share the same features into a higher-level entity set.

Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the sameway.

The terms specialization and generalization are used interchangeably.

Can have multiple specializations of an entity set based on different features.

E.g. *permanent_employee* vs. *temporary_employee*, in addition to
officer vs. *secretary* vs. *teller*

Each particular employee would be

a. a member of one of *permanent_employee* or *temporary_employee*,

b. and also a member of one of *officer*, *secretary*, or *teller*

The ISA relationship also referred to as **superclass – subclass** relationship

UNIT II

Relational Database Design:

- **Features of good** Relational designs
- Atomic domains and First Normal Form
- Decomposition using functional dependencies
- Functional dependency theory
- Decomposition using functional
- Decomposition using multivalued dependencies
- More Normal forms

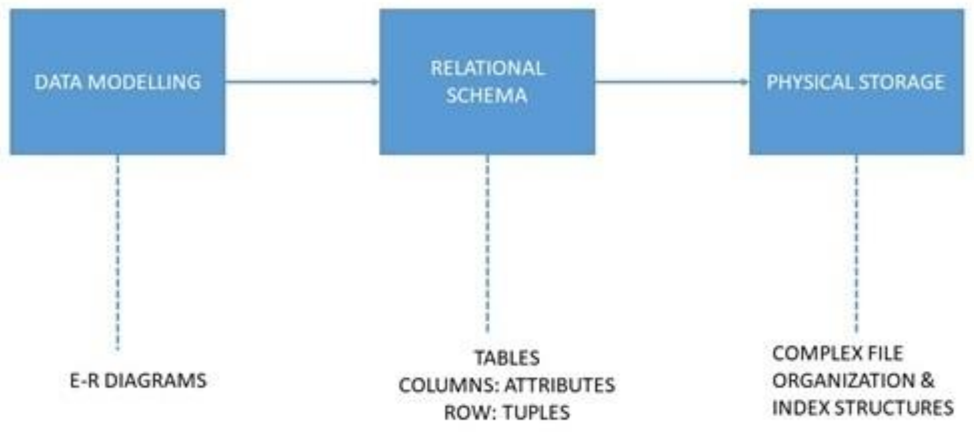
Database design process:

- Modeling temporal data

INTRODUCTION

The relational data model was introduced by C. F. Codd in 1970. Currently, it is the most widely used data model. The relational data model describes the world as “a collection of inter-related relations (or tables).” A relational data model involves the use of data tables that collect groups of elements into relations. These models work based on the idea that each table setup will include a primary key or identifier. Other tables use that identifier to provide "relational" data links and results.

Today, there are many commercial Relational Database Management System (RDBMS), such as Oracle, IBM DB2, and Microsoft SQL Server. There are also many free and open-source RDBMS, such as MySQL, mSQL (mini-SQL) and the embedded Java DB (Apache Derby). Database administrators use Structured Query Language (SQL) to retrieve data elements from a relational database.



As mentioned, the primary key is a fundamental tool in creating and using relational data models. It must be unique for each member of a data set. It must be populated for all members. Inconsistencies can cause problems in how developers retrieve data. Other issues with relational database designs include excessive duplication of data, faulty or partial data, or improper links or associations between tables. A large part of routine database administration involves evaluating all the data sets in a database to make sure that they are consistently populated and will respond well to SQL or any other data retrieval method.

Database Design Objective

- **Eliminate Data Redundancy:** the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- **Ensure Data Integrity and Accuracy:** is the maintenance of, and the assurance of the accuracy and consistency of, data over its entire life-cycle, and is a critical aspect to the design, implementation, and usage of any system which stores, processes, or retrieves data.

The relational model has provided the basis for:

- Research on the theory of data/relationship/constraint
- Numerous database design methodologies
- The standard database access language called structured query language (SQL)
- Almost all modern commercial database management systems

Relational databases go together with the development of SQL. The simplicity of SQL - where even a novice can learn to perform basic queries in a short period of time - is a large part of the reason for the popularity of the relational model.

The two tables below relate to each other through the product code field. Any two tables can relate to each other simply by creating a field they have in common.

Table 1

Product_code	**Description**	**Price**
A416	Colour Pen	₹ 25.00
C923	Pencil box	₹ 45.00

Table 2

Invoice_code	**Invoice_line**	**Product_code**	**Quantity**
3804	1	A416	15
3804	2	C923	24

There are four stages of an RDM which are as follows –

- **Relations and attributes –** The various tables and attributes related to each table are identified. The tables represent entities, and the attributes represent the properties of the respective entities.
- **Primary keys –** The attribute or set of attributes that help in uniquely identifying a record is identified and assigned as the primary key.
- **Relationships –**The relationships between the various tables are established with the help of foreign keys. Foreign keys are attributes occurring in a table that are primary keys of another table. The types of relationships that can exist between the relations (tables) are One to one, One to many, and Many to many
- **Normalization –** This is the process of optimizing the database structure. Normalization simplifies the database design to avoid redundancy and confusion. The different normal forms are as follows:

  1. First Normal form
  2. Second normal form
  3. Third normal form
  4. Boyce-Codd normal form
  5. Fifth normal form

By applying a set of rules, a table is normalized into the above normal forms in a linearly progressive fashion. The efficiency of the design gets better with each higher degree of normalization.

Advantages of Relational Databases

The main advantages of relational databases are that they enable users to easily categorize and store data that can later be queried and filtered to extract specific information for reports. Relational databases are also easy to extend and aren't reliant on the physical organization. After the original database creation, a new data category can be added without all existing applications being modified.

Other Advantages

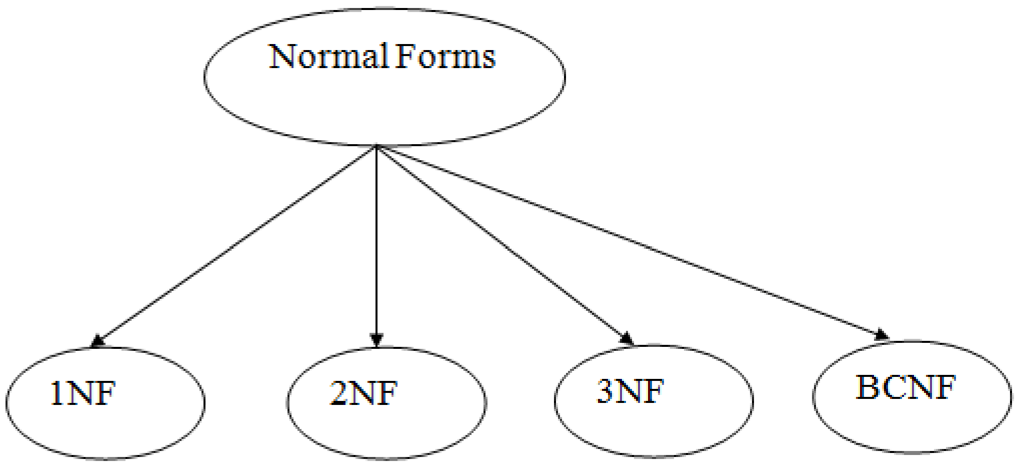
- **Accurate –** Data is stored just once, which eliminates data deduplication.
- **Flexible –** Complex queries are easy for users to carry out.
- **Collaborative –**Multiple users can access the same database.
- **Trusted –**Relational database models are mature and well-understood.
- **Secure –** Data in tables within relational database management systems (RDBMS) can be limited to allow access by only particular users.

Normalization

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

Types of Normal Forms

There are the four types of normal forms:



Insertion Anomaly

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

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

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

Updation Anomaly

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

Deletion Anomaly

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

<u>Normalization Rule</u>

Normalization rules are divided into the following normal forms:

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

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

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

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

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

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

For a table to be in the Second Normal Form,

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

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

<u>Third Normal Form (3NF)</u>

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

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

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

Boyce and Codd Normal Form (BCNF)

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

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

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

Fourth Normal Form (4NF)

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

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

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

What is First Normal Form (1NF)?

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

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

Rules for First Normal Form

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

Rule 1: Single Valued Attributes

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

Rule 2: Attribute Domain should not change

This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

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

Rule 3: Unique name for Attributes/Columns

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

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

Rule 4: Order doesn't matters

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

Multi-valued dependency, Theory and Decomposition

When existence of one or more rows in a table implies one or more other rows in the same table, then the Multi-valued dependencies occur.

If a table has attributes P, Q and R, then Q and R are multi-valued facts of P.

It is represented by double arrow –

```
->->
```

Example

The following is an example that would make it easier to understand functional dependency –

We have a **<Department>** table with two attributes – **DeptId** and **DeptName**.

DeptId = Department ID **DeptName** = Department Name

The **DeptId** is our primary key. Here, **DeptId** uniquely identifies the **DeptName** attribute. This is because if you want to know the department name, then at first you need to have the **DeptId**.

DeptId	**DeptName**
001	Finance
002	Marketing
003	HR

Therefore, the above functional dependency between **DeptId** and **DeptName** can be determined as **DeptId** is functionally dependent on **DeptName** –

DeptId -> DeptName

Types of Functional Dependency

Functional Dependency has three forms –

- Trivial Functional Dependency
- Non-Trivial Functional Dependency
- Completely Non-Trivial Functional Dependency

Let us begin with Trivial Functional Dependency –

Trivial Functional Dependency

It occurs when B is a subset of A in –

```
A ->B
```

Example

We are considering the same **<Department>** table with two attributes to understand the concept of trivial dependency.

The following is a trivial functional dependency since **DeptId** is a subset of **DeptId** and **DeptName**

```
{ DeptId,  DeptName } -> Dept Id
```

Non –Trivial Functional Dependency

It occurs when B is not a subset of A in –

```
A ->B
```

Example

```
DeptId ->  DeptName
```

The above is a non-trivial functional dependency since DeptName is a not a subset of DeptId.

Completely Non - Trivial Functional Dependency

It occurs when A intersection B is null in –

```
A ->B
```

Armstrong's Axioms Property of Functional Dependency

Armstrong's Axioms property was developed by William Armstrong in 1974 to reason about functional dependencies.

The property suggests rules that hold true if the following are satisfied:

- **Transitivity**
  If A->B and B->C, then A->C i.e. a transitive relation.
- **Reflexivity**
  A-> B, if B is a subset of A.
- **Augmentation**
  The last rule suggests: AC->BC, if A->B

MORE NORMAL FORMS

SOME FACTS ABOUT DATABASE NORMALIZATION

- The words normalization and normal form refer to the structure of a database.
- Normalization was developed by IBM researcher E.F. Codd In the 1970s.
- Normalization increases clarity in organizing data in Databases.

Normalization of a Database is achieved by following a set of rules called 'forms' in creating the database.

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.

- In the second normal form, all non-key attributes are fully functional dependent on the primary key

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

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

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

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

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

TEACHER_SUBJECT table:

Third Normal Form (3NF)

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

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

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

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP

222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	**EMP_STATE**	**EMP_CITY**
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF)

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

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

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

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

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

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

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India

264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

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

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

Fourth normal form (4NF)

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

Example

STUDENT

STU_ID	**COURSE**	**HOBBY**
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

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

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

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

STUDENT_COURSE

STU_ID	**COURSE**
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	**HOBBY**
21	Dancing
21	Singing
34	Dancing
74	Cricket

59	Hockey

Fifth normal form (5NF)

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

Example

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

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

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

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

P1

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

P2

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

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

DATABASE DESIGN PROCESS

Database Design is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems. Properly designed database are easy to maintain, improves data consistency and are cost effective in terms of disk storage space. The database designer decides how the data elements correlate and what data must be stored.

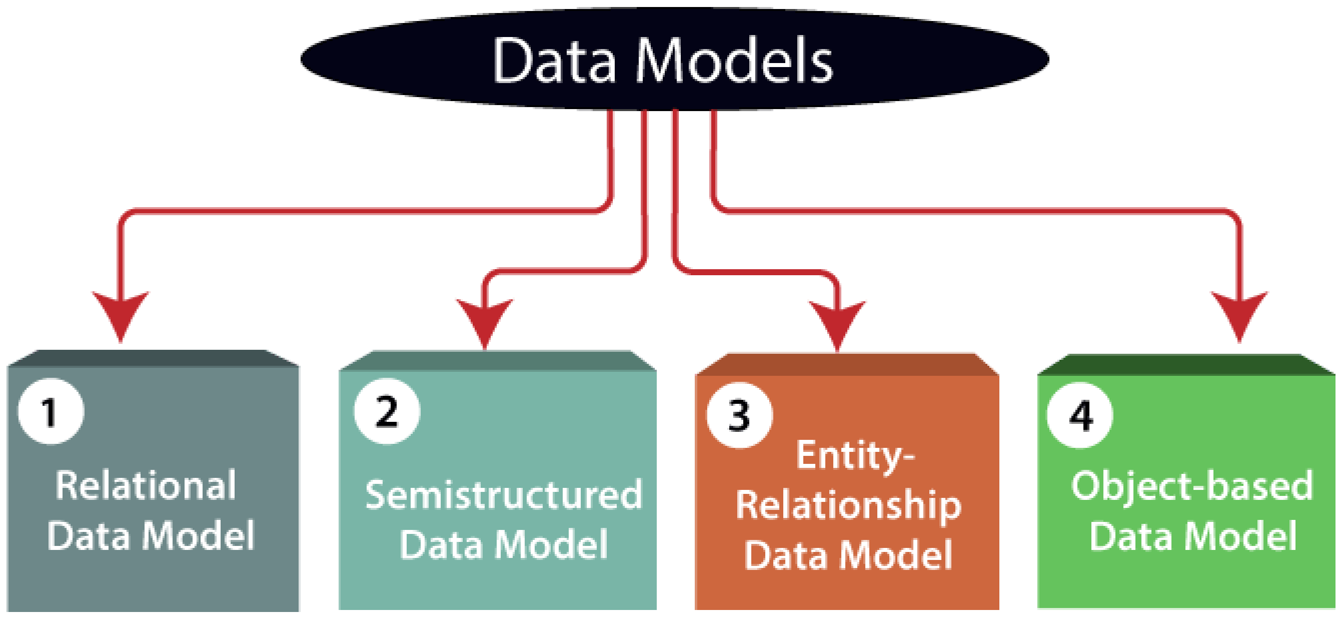
The main objectives of database designing are to produce logical and physical designs models of the proposed database system.

The logical model concentrates on the data requirements and the data to be stored independent of physical considerations. It does not concern itself with how the data will be stored or where it will be stored physically.

The physical data design model involves translating the logical design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

Data Models

Data Model is the modeling of the data description, data semantics, and consistency constraints of the data. It provides the conceptual tools for describing the design of a database at each level of data abstraction. Therefore, there are following four data models used for understanding the structure of the database:


Data Models
1 Relational Data Model
2 Semistructured Data Model
3 Entity-Relationship Data Model
4 Object-based Data Model

1) Relational Data Model: This type of model designs the data in the form of rows and columns within a table. Thus, a relational model uses tables for representing data and in-between relationships. Tables are also called relations. This model was initially described by Edgar F. Codd, in 1969. The relational data model is the widely used model which is primarily used by commercial data processing applications.

2) Entity-Relationship Data Model: An ER model is the logical representation of data as objects and relationships among them. These objects are known as entities, and relationship is an association among these entities. This model was designed by Peter Chen and published in 1976 papers. It was widely used in database designing. A set of attributes describe the entities. For example, student_name, student_id describes the 'student' entity. A set of the same type of entities is known as an 'Entity set', and the set of the same type of relationships is known as 'relationship set'.

3) Object-based Data Model: An extension of the ER model with notions of functions, encapsulation, and object identity, as well. This model supports a rich type system that includes structured and collection types. Thus, in 1980s, various database systems following the object-oriented approach were developed. Here, the objects are nothing but the data carrying its properties.

4) Semistructured Data Model: This type of data model is different from the other three data models (explained above). The semistructured data model allows the data specifications at places where the individual data items of the same type may have different attributes sets. The Extensible Markup Language, also known as XML, is widely used for representing the semistructured data. Although XML was initially designed for including the markup information to the text document, it gains importance because of its application in the exchange of data.

TEMPORAL DATA

Temporal databases support managing and accessing temporal data by providing one or more of the following features:[1][2]

- A time period datatype, including the ability to represent time periods with no end (infinity or forever)
- The ability to define valid and transaction time period attributes and bitemporal relations
- System-maintained transaction time
- Temporal primary keys, including non-overlapping period constraints
- Temporal constraints, including non-overlapping uniqueness and referential integrity
- Update and deletion of temporal records with automatic splitting and coalescing of time periods
- Temporal queries at current time, time points in the past or future, or over durations
- Predicates for querying time periods, often based on Allen's interval relations

UNIT III

Database System Architecture:

- Centralized and Client-Server architecture
- Server system architecture
- Parallel systems
- Distributed systems

Network types. Parallel databases:

- I/O parallelism
- Interquery Parallelism

Distributed Databases:

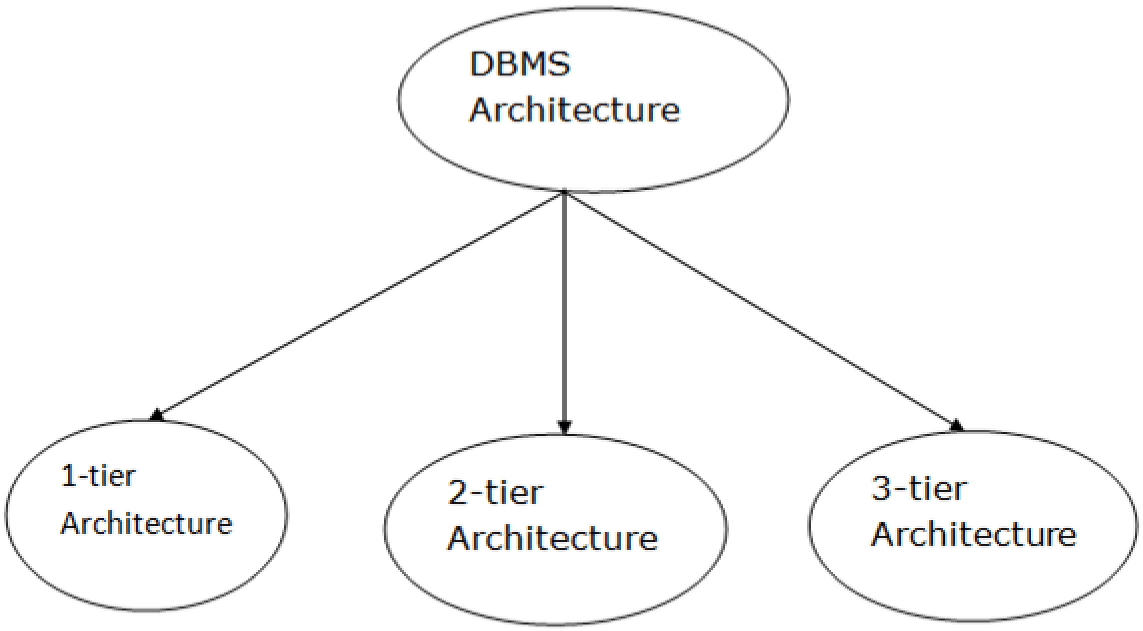
- Homogeneous and Heterogeneous databases
- Distributed Data storage
- Distributed transactions
- Distributed query processing.

Database System Architecture

<u>DBMS Architecture</u>

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

<u>Types of DBMS Architecture</u>



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 for end users.
- The 1-Tier architecture is used for development of the local application, where programmers can 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.

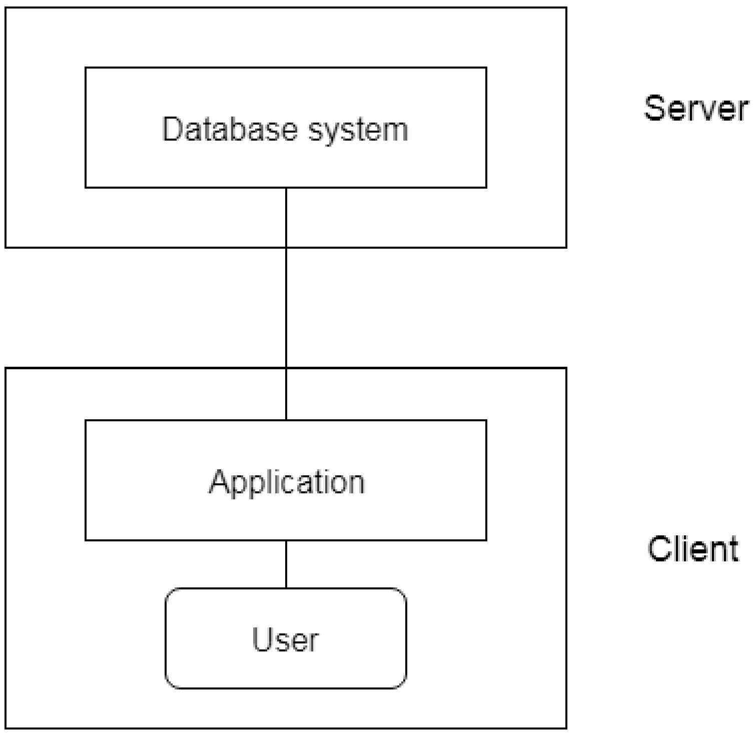


Fig: 2-tier Architecture

3-Tier Architecture

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

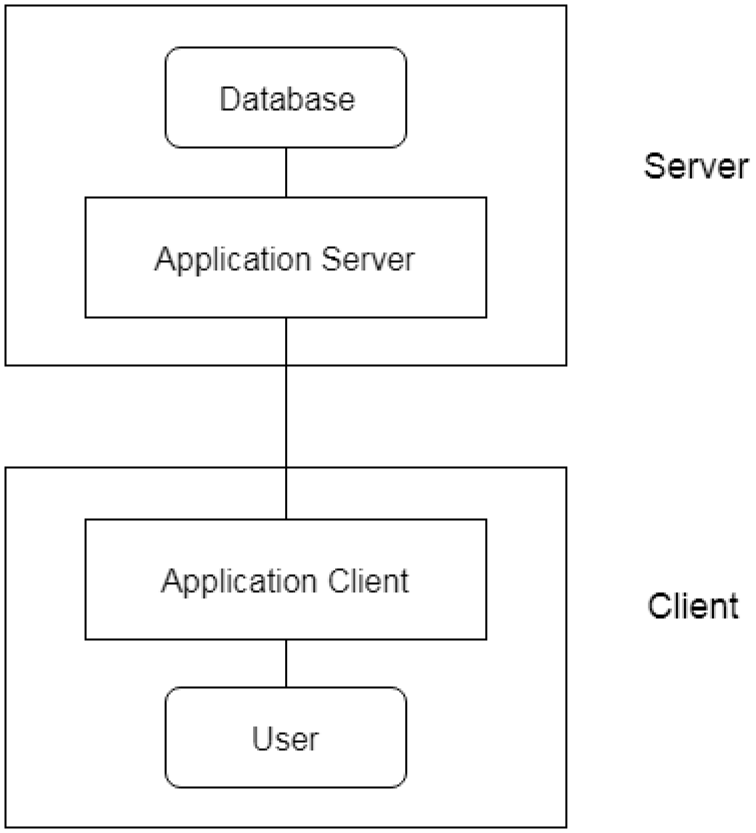


Fig: 3-tier Architecture

1. Centralized DBMS Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality, application program execution, and user inter-face processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

<u>2. Client/Server Architectures</u>

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers,

Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless work-stations or workstations/PCs with disks that have only client software installed).

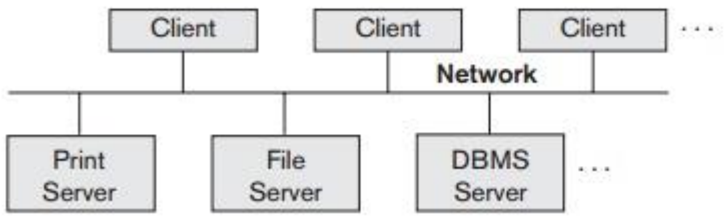


Figure 2.5
Logical two-tier client/server architecture.

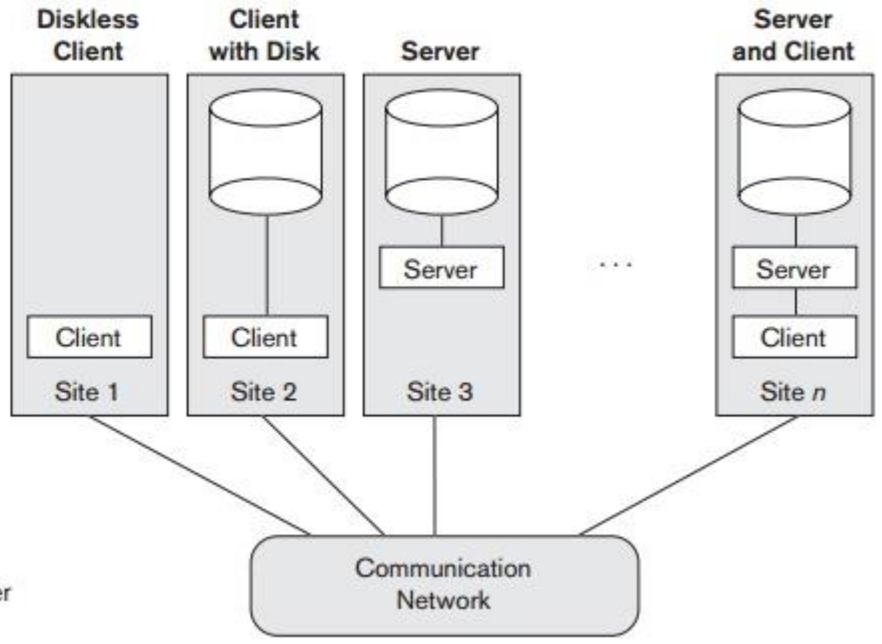


Figure 2.6
Physical two-tier client/server architecture.

Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality— such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hard-ware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate machines.

PARALLEL SYSTEMS

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

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

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

DISTRIBUTED SYSTEMS

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.

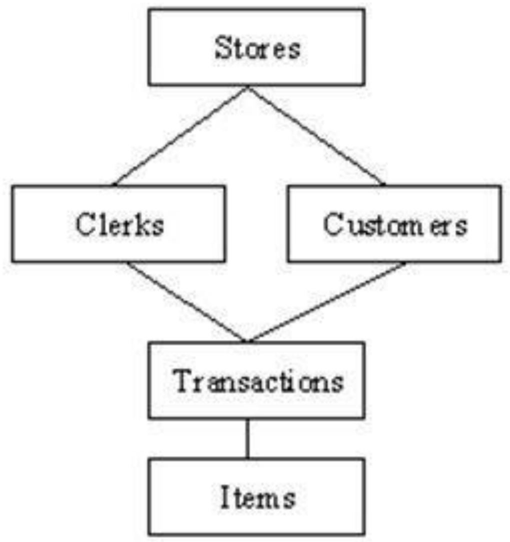
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

NETWORK TYPES

Network database management systems (Network DBMSs) are based on a network data model that allows each record to have multiple parents and multiple child records. A network database allows flexible relationship model between entities.

There are several types of database management systems such as relational, network, graph, and hierarchical.

The following diagram represents a network data model that shows that the Stores entity has relationship with multiple child entities and the Transactions entity has relationships with multiple parent entities. In other words, a network database model allows one parent to have multiple child record sets and each record set can be linked to multiple nodes (parents) and children.



The network model was developed and presented by Charles Bachman in 1969. The network model often used to build computer network systems and is an enhancement to the hierarchical database model. Learn more here - What are hierarchical databases

The key advantage of a network database model is its supports many-to-many relationship and hence provides greater flexibility and accessibility. The result is a faster data access, search, and navigation.

Some of the popular network databases are,

1. Integrated Data Store (IDS)
2. IDMS (Integrated Database Management System)
3. Raima Database Manager
4. TurboIMAGE
5. Univac DMS-1100

PARALLEL DATABASE

Introduction

• Parallel machines are becoming quite common and affordable – Prices of microprocessors, memory and disks have dropped sharply

• Databases are growing increasingly large – large volumes of transaction data are collected and stored for later analysis. – multimedia objects like images are increasingly stored in databases

• Large-scale parallel database systems increasingly used for: – processing time-consuming decision-support queries – providing high throughput for transaction processing.

Data can be partitioned across multiple disks for parallel I/O.

• Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel – data can be partitioned and each processor can work independently on its own partition.

• Queries are expressed in high level language (SQL, translated to relational algebra) – makes parallelization easier.

• Different queries can be run in parallel with each other. Concurrency control takes care of conflicts.

• Thus, databases naturally lend themselves to parallelism

I/O PARALLELISM

Data can be partitioned across multiple disks for parallel I/O.

Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.

• Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.

• Partitioning techniques (number of disks = n): Round-robin : Send the ith tuple inserted in the relation to disk i mod n.

Hash partitioning :

– Choose one or more attributes as the partitioning attributes.

– Choose hash function h with range 0 ...n − 1.

– Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to disk i

Range partitioning :

– Choose an attribute as the partitioning attribute.

– A partitioning vector [v0, v1,...,vn−2] is chosen

– Let v be the partitioning attribute value of a tuple.

Tuples such that vi ≤ v < vi+1 go to disk i + 1.

Tuples with v < v0 go to disk 0 and tuples with v ≥ vn−2 go to disk n − 1. E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk 2.

<u>Comparison of Partitioning Techniques</u>

Evaluate how well partitioning techniques support the following types of data access:

1. Scanning the entire relation.

2. Locating a tuple associatively – point queries. – E.g., r.A = 25.

3. Locating all tuples such that the value of a given attribute lies within a specified range – range queries. – E.g., 10 ≤ r.A < 25

Round-robin.

– Best suited for sequential scan of entire relation on each query.

∗ All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.

– Range queries are difficult to process

∗ No clustering – tuples are scattered across all disks

Hash partitioning.

– Good for sequential access

∗ Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks

∗ Retrieval work is then well balanced between disks.

– Good for point queries on partitioning attribute

∗ Can lookup single disk, leaving others available for answering other queries.

∗ Index on partitioning attribute can be local to disk, making lookup and update more efficient

– No clustering, so difficult to answer range queries

Range partitioning.

– Provides data clustering by partitioning attribute value.

– Good for sequential access

- Good for point queries on partitioning attribute: only one disk needs to be accessed.

– For range queries on partitioning attribute, one to a few disks may need to be accessed

∗ Remaining disks are available for other queries.

∗ Good if result tuples are from one to a few blocks.

∗ If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted · Example of execution skew.

Interquery Parallelism

Queries/transactions execute in parallel with one another.

• Increases transaction throughput; used primarily to scale up a transaction processing system to support a larger number of transactions per second.

• Easiest form of parallelism to support, particularly in a shared-memory parallel database, because even sequential database systems support concurrent processing.

• More complicated to implement on shared-disk or shared-nothing architectures

– Locking and logging must be coordinated by passing messages between processors.

– Data in a local buffer may have been updated at another processor.
– Cache-coherency has to be maintained
- reads and writes of data in buffer must find latest version of data.

Intraquery Parallelism

Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
• Two complementary forms of intraquery parallelism :
Intraoperation Parallelism
– parallelize the execution of each individual operation in the query. – Interoperation Parallelism
– execute the different operations in a query expression in parallel. the first form scales better with increasing parallelism because the number of tuples processed by each operation is typically more than the number of operations in a query

DISTRIBUTED DATABASES:

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.

- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Distributed Database Management System

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

Factors Encouraging DDBMS

The following factors encourage moving over to DDBMS –

- **Distributed Nature of Organizational Units –** Most organizations in the current times are subdivided into multiple units that are physically distributed over the globe. Each unit requires its own set of local data. Thus, the overall database of the organization becomes distributed.
- **Need for Sharing of Data –** The multiple organizational units often need to communicate with each other and share their data and resources. This demands common databases or replicated databases that should be used in a synchronized manner.

- **Support for Both OLTP and OLAP** – Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) work upon diversified systems which may have common data. Distributed database systems aid both these processing by providing synchronized data.
- **Database Recovery** – One of the common techniques used in DDBMS is replication of data across different sites. Replication of data automatically helps in data recovery if database in any site is damaged. Users can access data from other sites while the damaged site is being reconstructed. Thus, database failure may become almost inconspicuous to users.
- **Support for Multiple Application Software** – Most organizations use a variety of application software each with its specific database support. DDBMS provides a uniform functionality for using the same data among different platforms.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development – If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

More Reliable – In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response – If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost – In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

Adversities of Distributed Databases

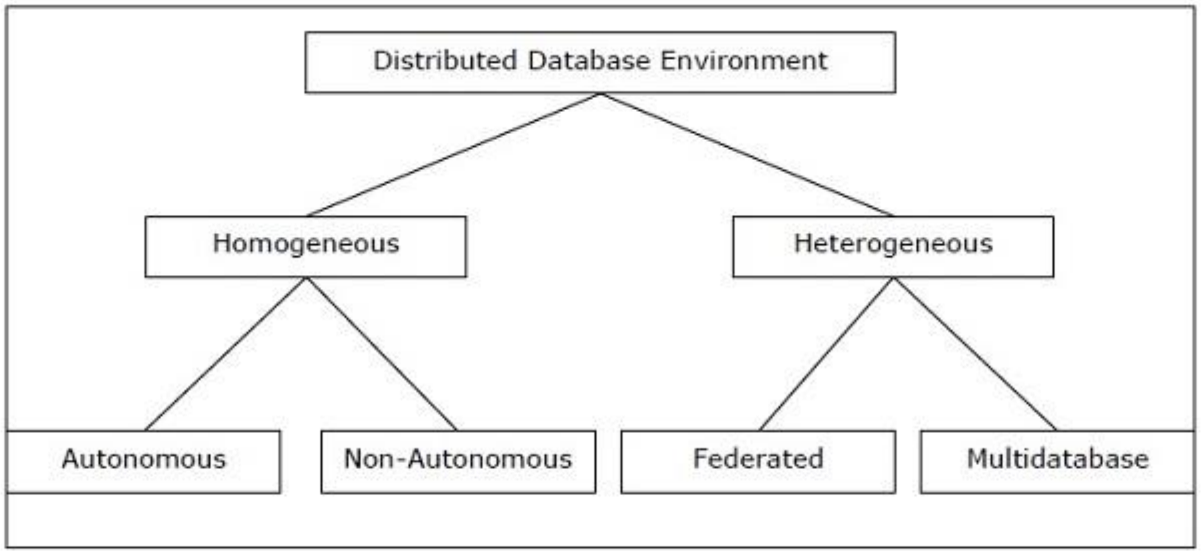
Following are some of the adversities associated with distributed databases.

- **Need for complex and expensive software –** DDBMS demands complex and often expensive software to provide data transparency and co-ordination across the several sites.
- **Processing overhead –** Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Data integrity –** The need for updating data in multiple sites pose problems of data integrity.
- **Overheads for improper data distribution –** Responsiveness of queries is largely dependent upon proper data distribution. Improper data distribution often leads to very slow response to user requests.

Homogeneous and Heterogeneous databases

Types of Distributed Databases

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further sub-divisions, as shown in the following illustration.



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are –

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database –

- **Autonomous** – Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.
- **Non-autonomous** – Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are −

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- **Federated** − The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Un-federated** − The database systems employ a central coordinating module through which the databases are accessed.

Distributed DBMS Architectures

DDBMS architectures are generally developed depending on three parameters −

- **Distribution** − It states the physical distribution of data across the different sites.
- **Autonomy** − It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.
- **Heterogeneity** − It refers to the uniformity or dissimilarity of the data models, system components and databases.

Architectural Models

Some of the common architectural models are −

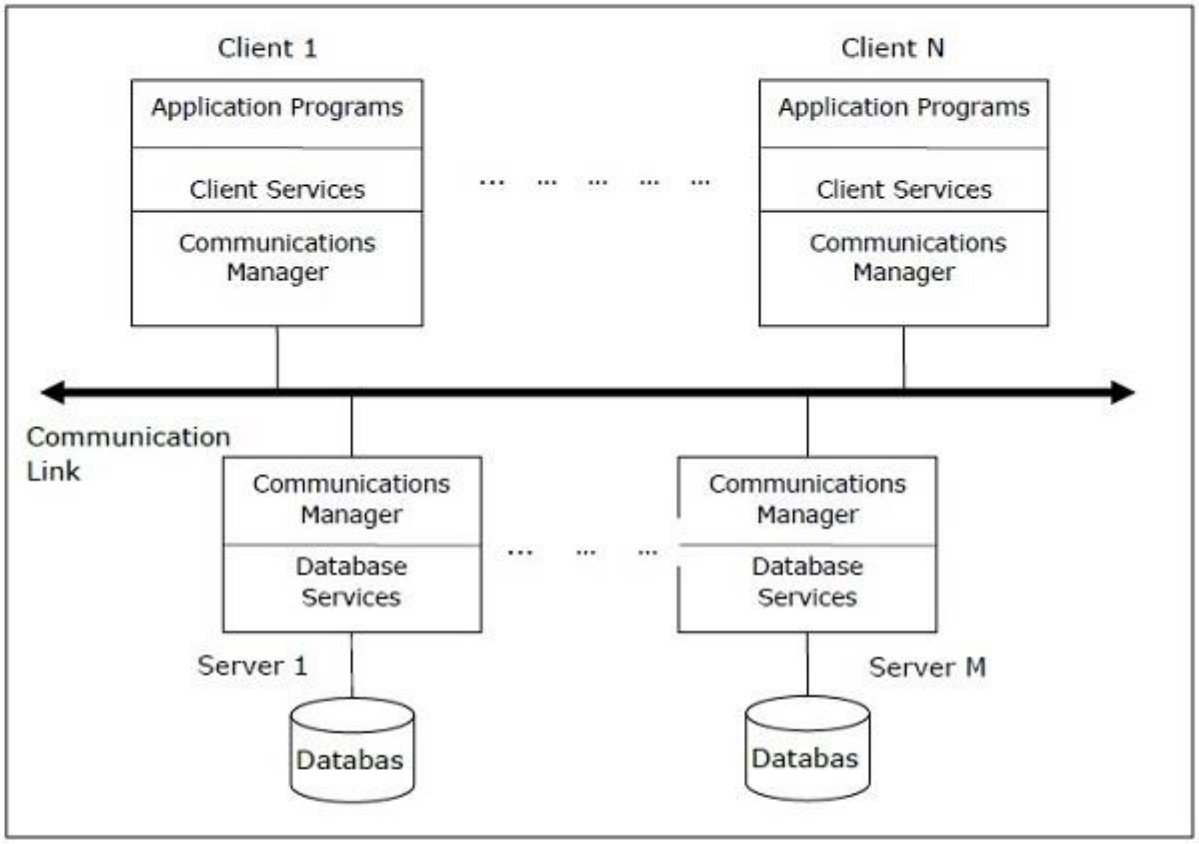
- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

Client - Server Architecture for DDBMS

This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.

The two different client - server architecture are −

- Single Server Multiple Client
- Multiple Server Multiple Client (shown in the following diagram)

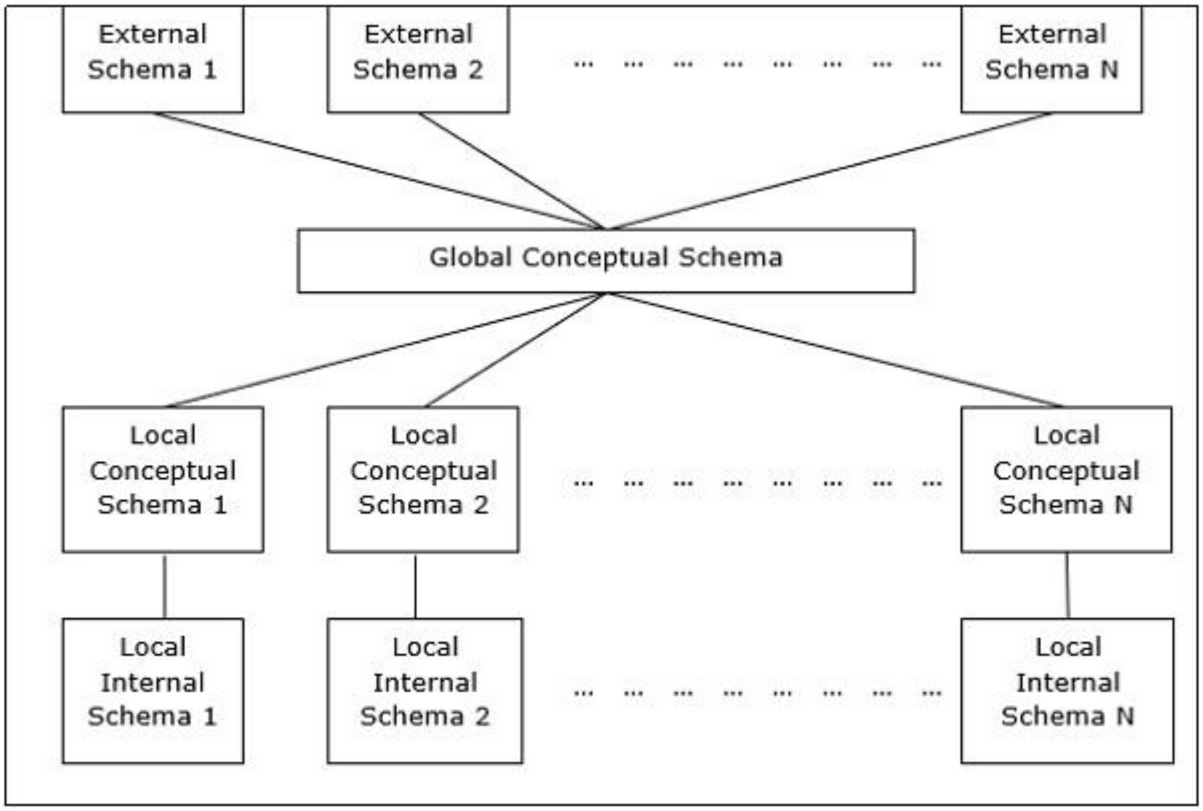


Peer- to-Peer Architecture for DDBMS

In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.

This architecture generally has four levels of schemas –

- **Global Conceptual Schema** – Depicts the global logical view of data.
- **Local Conceptual Schema** – Depicts logical data organization at each site.
- **Local Internal Schema** – Depicts physical data organization at each site.
- **External Schema** – Depicts user view of data.



Multi - DBMS Architectures

This is an integrated database system formed by a collection of two or more autonomous database systems.

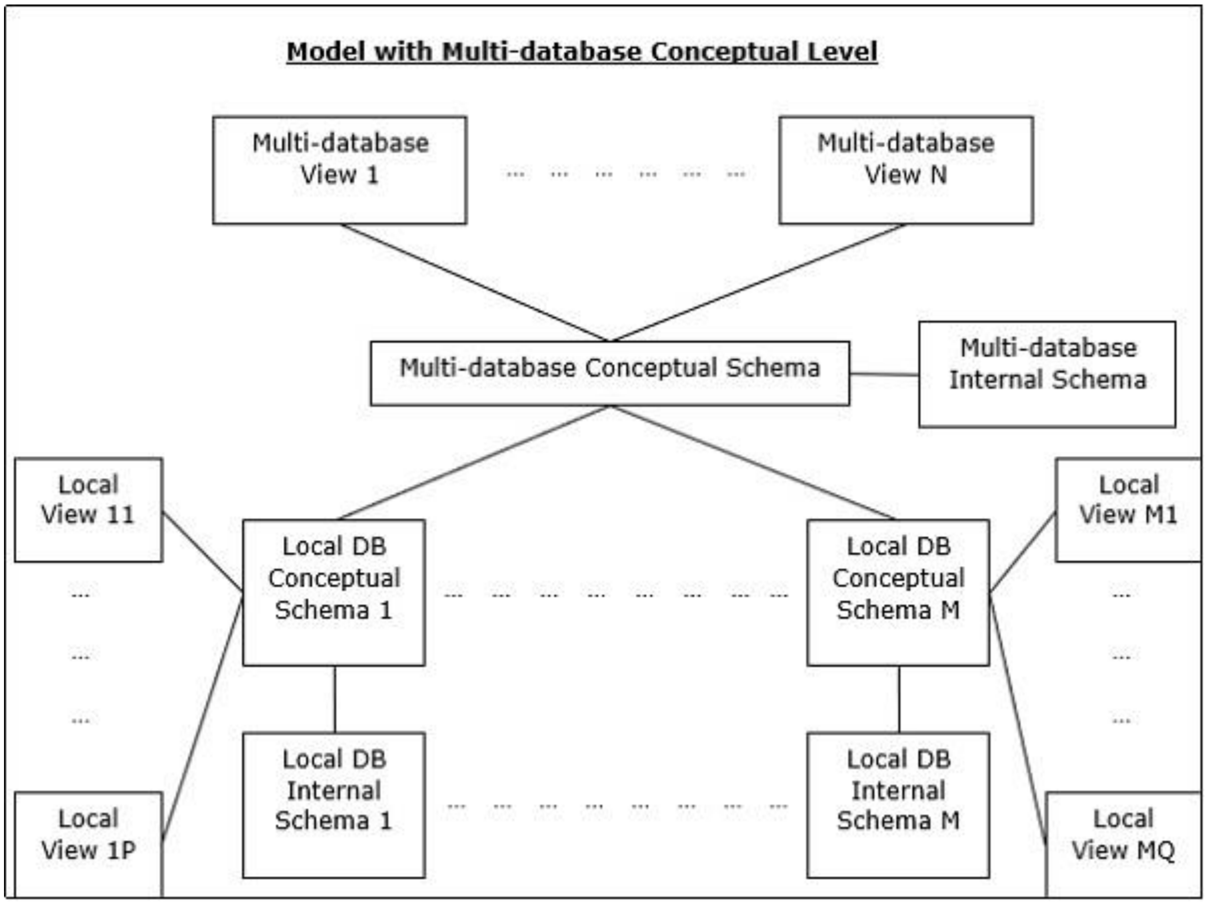
Multi-DBMS can be expressed through six levels of schemas –

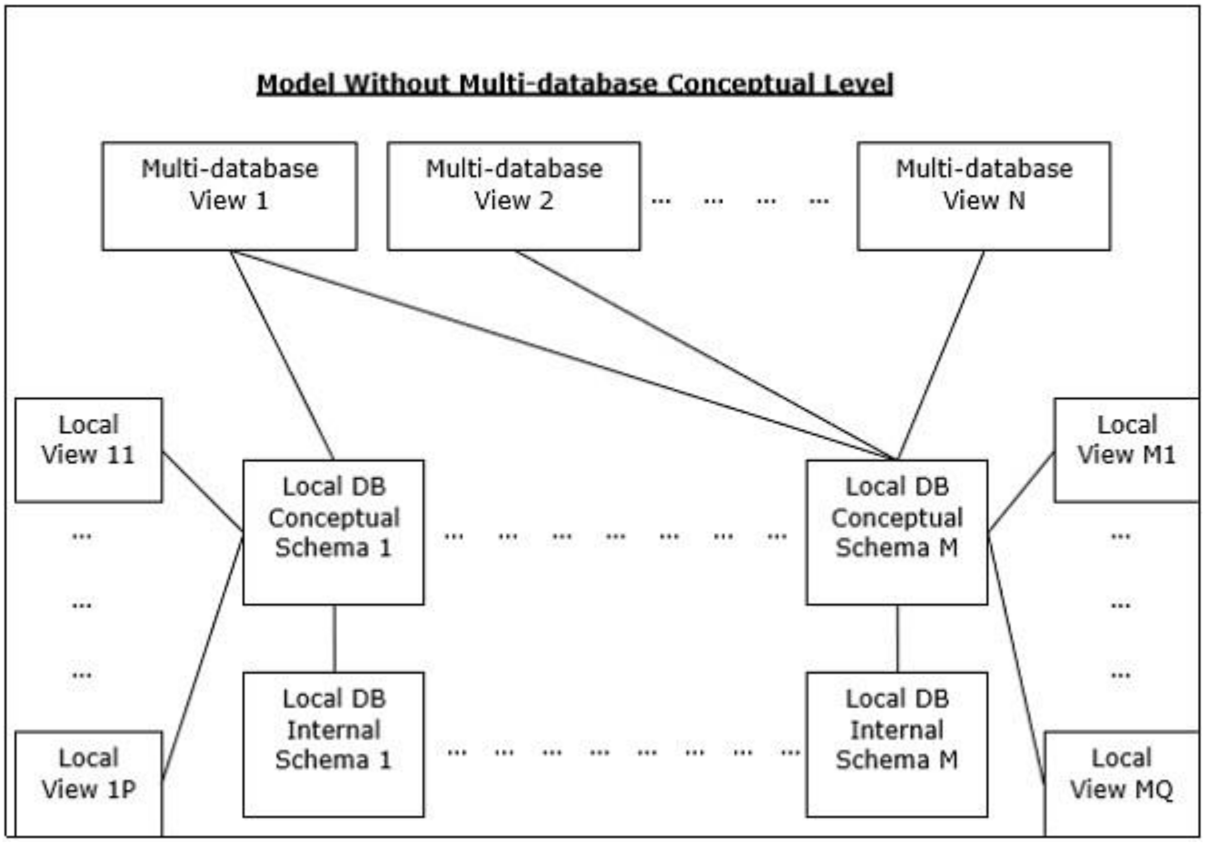
- **Multi-database View Level –** Depicts multiple user views comprising of subsets of the integrated distributed database.
- **Multi-database Conceptual Level –** Depicts integrated multi-database that comprises of global logical multi-database structure definitions.
- **Multi-database Internal Level –** Depicts the data distribution across different sites and multi-database to local data mapping.
- **Local database View Level –** Depicts public view of local data.
- **Local database Conceptual Level –** Depicts local data organization at each site.

- **Local database Internal Level –** Depicts physical data organization at each site.

There are two design alternatives for multi-DBMS –

- Model with multi-database conceptual level.
- Model without multi-database conceptual level.


Model with Multi-database Conceptual Level
Multi-database View 1
Multi-database View N
Multi-database Conceptual Schema
Multi-database Internal Schema
Local View 11
Local View 1P
Local DB Conceptual Schema 1
Local DB Conceptual Schema M
Local DB Internal Schema 1
Local DB Internal Schema M
Local View M1
Local View MQ


Model Without Multi-database Conceptual Level
Multi-database View 1
Multi-database View 2
Multi-database View N
Local View 11
Local View 1P
Local DB Conceptual Schema 1
Local DB Conceptual Schema M
Local DB Internal Schema 1
Local DB Internal Schema M
Local View M1
Local View MQ

Design Alternatives

The distribution design alternatives for the tables in a DDBMS are as follows –

- Non-replicated and non-fragmented
- Fully replicated
- Partially replicated
- Fragmented
- Mixed

Non-replicated & Non-fragmented

In this design alternative, different tables are placed at different sites. Data is placed so that it is at a close proximity to the site where it is used most. It is most suitable for database systems where the percentage of queries needed to join information in tables placed at different sites is low. If an appropriate distribution strategy is adopted, then this design alternative helps to reduce the communication cost during data processing.

Fully Replicated

In this design alternative, at each site, one copy of all the database tables is stored. Since, each site has its own copy of the entire database, queries are very fast requiring negligible communication cost. On the contrary, the massive redundancy in data requires huge cost during update operations. Hence, this is suitable for systems where a large number of queries is required to be handled whereas the number of database updates is low.

Partially Replicated

Copies of tables or portions of tables are stored at different sites. The distribution of the tables is done in accordance to the frequency of access. This takes into consideration the fact that the frequency of accessing the tables vary considerably from site to site. The number of copies of the tables (or portions) depends on how frequently the access queries execute and the site which generate the access queries.

Fragmented

In this design, a table is divided into two or more pieces referred to as fragments or partitions, and each fragment can be stored at different sites. This considers the fact that it seldom happens that all data stored in a table is required at a given site. Moreover, fragmentation increases parallelism and provides better disaster recovery. Here, there is only one copy of each fragment in the system, i.e. no redundant data.

The three fragmentation techniques are −

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

Mixed Distribution

This is a combination of fragmentation and partial replications. Here, the tables are initially fragmented in any form (horizontal or vertical), and then these fragments are partially replicated across the different sites according to the frequency of accessing the fragments.

DISTRIBUTED TRANSACTIONS

Distributed transaction management deals with the problems of always providing a consistent distributed database in the presence of a large number of transactions (local and global) and failures (communication link and/or site failures). This is accomplished through

(i) distributed commit protocols that guarantee atomicity property;

(ii) distributed concurrency control techniques to ensure consistency and isolation properties; and

iii) distributed recovery methods to preserve consistency and durability when failures occur.

A transaction is a sequence of actions on a database that forms a basic unit of reliable and consistent computing, and satisfies the ACID property. In a distributed database system (DDBS), transactions may be local or global. In local transactions, the actions access and update data in a single site only, and hence it is straightforward to ensure the ACID property.

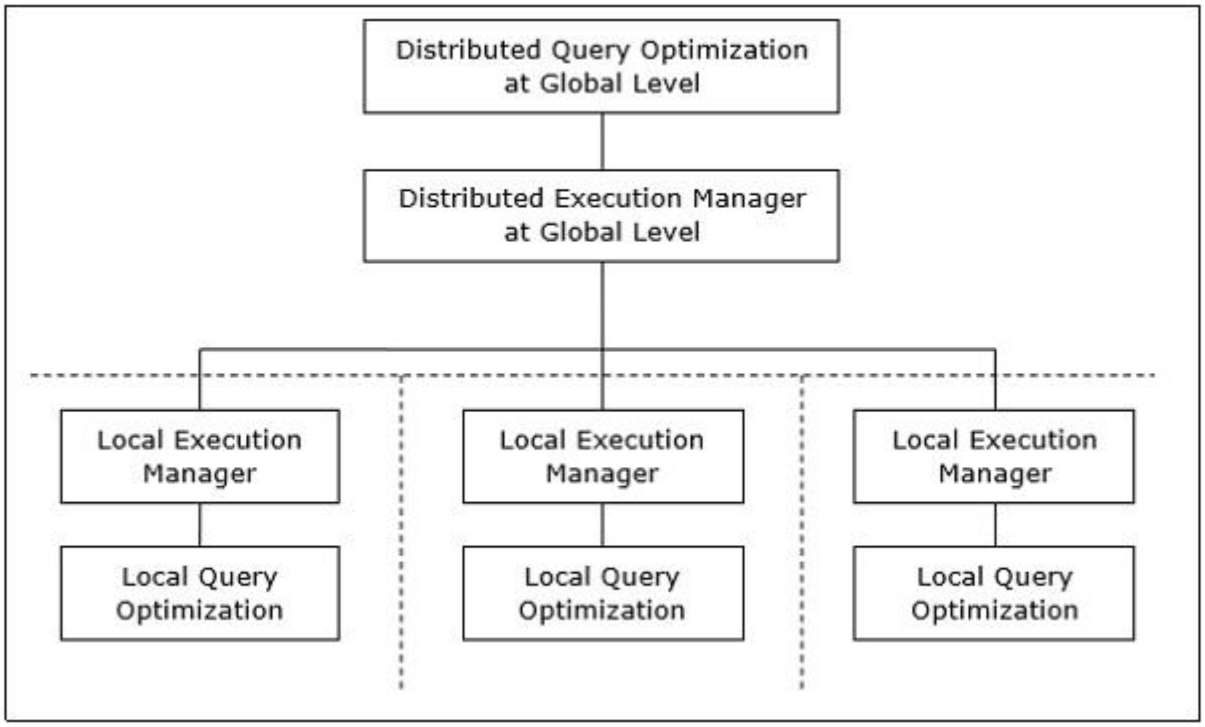
DISTRIBUTED QUERY PROCESSING

Distributed query processing is the procedure of answering queries (which means mainly read operations on large data sets) in a distributed environment where data is managed at multiple sites in a computer network. Query processing involves the transformation of a high-level query (e.g., formulated in SQL) into a query execution plan (consisting of lower-level query operators in some variation of relational algebra) as well as the execution of this plan. The goal of the transformation is to produce a plan which is equivalent to the original query (returning the same result) and efficient, i.e., to minimize resource consumption like total costs or response time.

Distributed Query Processing Architecture

In a distributed database system, processing a query comprises of optimization at both the global and the local level. The query enters the database system at the client or controlling site. Here, the user is validated, the query is checked, translated, and optimized at a global level.

The architecture can be represented as –



Mapping Global Queries into Local Queries

The process of mapping global queries to local ones can be realized as follows –

- The tables required in a global query have fragments distributed across multiple sites. The local databases have information only about local data. The controlling site uses the global data dictionary to gather information about the distribution and reconstructs the global view from the fragments.

- If there is no replication, the global optimizer runs local queries at the sites where the fragments are stored. If there is replication, the global optimizer selects the site based upon communication cost, workload, and server speed.
- The global optimizer generates a distributed execution plan so that least amount of data transfer occurs across the sites. The plan states the location of the fragments, order in which query steps needs to be executed and the processes involved in transferring intermediate results.
- The local queries are optimized by the local database servers. Finally, the local query results are merged together through union operation in case of horizontal fragments and join operation for vertical fragments.

For example, let us consider that the following Project schema is horizontally fragmented according to City, the cities being New Delhi, Kolkata and Hyderabad.

Distributed Query Optimization

Distributed query optimization requires evaluation of a large number of query trees each of which produce the required results of a query. This is primarily due to the presence of large amount of replicated and fragmented data. Hence, the target is to find an optimal solution instead of the best solution.

The main issues for distributed query optimization are –

- Optimal utilization of resources in the distributed system.
- Query trading.
- Reduction of solution space of the query.

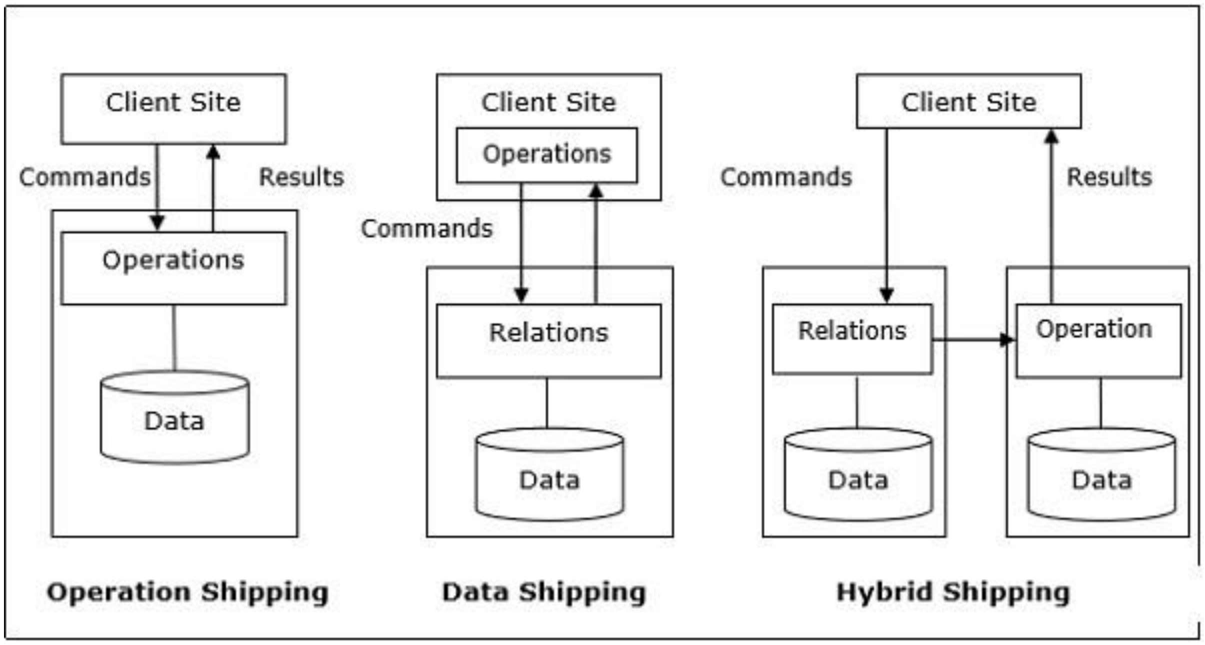
Optimal Utilization of Resources in the Distributed System

A distributed system has a number of database servers in the various sites to perform the operations pertaining to a query. Following are the approaches for optimal resource utilization –

Operation Shipping – In operation shipping, the operation is run at the site where the data is stored and not at the client site. The results are then transferred to the client site. This is appropriate for operations where the operands are available at the same site. Example: Select and Project operations.

Data Shipping – In data shipping, the data fragments are transferred to the database server, where the operations are executed. This is used in operations where the operands are distributed at different sites. This is also appropriate in systems where the communication costs are low, and local processors are much slower than the client server.

Hybrid Shipping – This is a combination of data and operation shipping. Here, data fragments are transferred to the high-speed processors, where the operation runs. The results are then sent to the client site.



Query Trading

In query trading algorithm for distributed database systems, the controlling/client site for a distributed query is called the buyer and the sites where the local queries execute are called sellers. The buyer formulates a number of alternatives for choosing sellers and for reconstructing the global results. The target of the buyer is to achieve the optimal cost.

The algorithm starts with the buyer assigning sub-queries to the seller sites. The optimal plan is created from local optimized query plans proposed by the sellers combined with the communication cost for reconstructing the final result. Once the global optimal plan is formulated, the query is executed.

Reduction of Solution Space of the Query

Optimal solution generally involves reduction of solution space so that the cost of query and data transfer is reduced. This can be achieved through a set of heuristic rules, just as heuristics in centralized systems.

Following are some of the rules –

- Perform selection and projection operations as early as possible. This reduces the data flow over communication network.
- Simplify operations on horizontal fragments by eliminating selection conditions which are not relevant to a particular site.
- In case of join and union operations comprising of fragments located in multiple sites, transfer fragmented data to the site where most of the data is present and perform operation there.
- Use semi-join operation to qualify tuples that are to be joined. This reduces the amount of data transfer which in turn reduces communication cost.
- Merge the common leaves and sub-trees in a distributed query tree.

UNIT IV

1. DATA INTEGRITY
2. CREATING AND MAINTAINING TABLES
3. INDEXES
4. SEQUENCES
5. VIEWS
6. USERS, PRIVILEGES, AND ROLES
7. SYNONYMS

DATA INTEGRITY

Data **integrity** is normally enforced in a database system by a series of **integrity** constraints or **rules**.

Create table of employees:

```
DROP TABLE emp_tab;

CREATE TABLE emp_tab (

  empname VARCHAR2(80),

  empno   INTEGER,

  deptno  INTEGER

);
```

Create constraint to enforce rule that all values in department table are unique:

```
ALTER TABLE dept_tab ADD PRIMARY KEY (deptno);
```

Create constraint to enforce rule that every employee must work for a valid department:

```
ALTER TABLE emp_tab ADD FOREIGN KEY (deptno) REFERENCES dept_tab(deptno);
```

Now, whenever you insert an employee record into **emp_tab**, Oracle Database checks that its **deptno** value appears in **dept_tab**

NOT NULL Constraints

Example 5-3 Inserting NULL Values into Columns with NOT NULL Constraints

```
DESCRIBE DEPARTMENTS;
```

Result:

```
 Name                        Null?    Type
```

```
--------------------------------------- -------- ------------

 DEPARTMENT_ID                  NOT NULL NUMBER(4)

DEPARTMENT_NAME                   NOT NULL VARCHAR2(30)

MANAGER_ID                        NUMBER(6)

LOCATION_ID                       NUMBER(4)
```

Try to insert NULL into DEPARTMENT_ID column:

```
INSERT INTO DEPARTMENTS ( DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID,
LOCATION_ID)VALUES (NULL, 'Sales', 200, 1700);
```

Result:

```
VALUES (NULL, 'Sales', 200, 1700)

     *

ERROR at line 4:

ORA-01400: cannot insert NULL into ("HR"."DEPARTMENTS"."DEPARTMENT_ID")
```

PRIMARY KEY Constraint

The primary key of a table uniquely identifies each row and ensures that no duplicate rows exist (and typically, this is its only purpose). Therefore, a primary key value cannot be **NULL.**

A table can have at most one primary key, but that key can have multiple columns (that is, it can be a composite key). To designate a primary key, use the **PRIMARY KEY** constraint.

UNIQUE Constraints

Use a **UNIQUE** constraint (which designates a unique key) on any column or combination of columns (except the primary key) where duplicate non-**NULL** values are not allowed. For example:

shows a table with a **UNIQUE** constraint, a row that violates the constraint, and a row that satisfies it.

Figure 5-1 Rows That Violate and Satisfy a UNIQUE Constraint

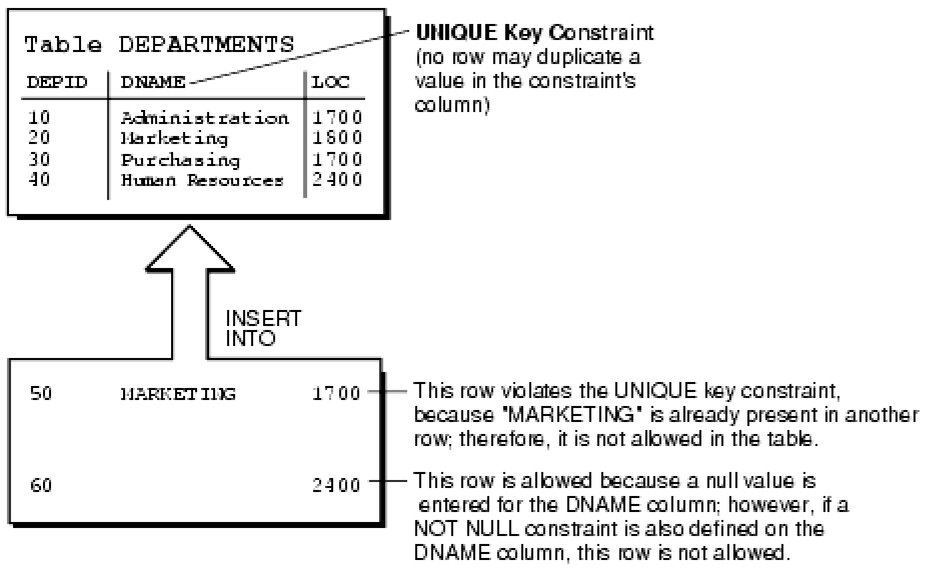


Table DEPARTMENTS
DEPID
DNAME
LOC
10 Administration 1700
20 Marketing 1800
30 Purchasing 1700
40 Human Resources 2400
UNIQUE Key Constraint (no row may duplicate a value in the constraint's column)
INSERT INTO
50 MARKETING 1700
60 2400
This row violates the UNIQUE key constraint, because "MARKETING" is already present in another row; therefore, it is not allowed in the table.
This row is allowed because a null value is entered for the DNAME column; however, if a NOT NULL constraint is also defined on the DNAME column, this row is not allowed.

FOREIGN KEY Constraints

When two tables share one or more columns, you use can use a **FOREIGN KEY** constraint to enforce referential integrity—that is, to ensure that the shared columns always have the same values in both tables.

Designate one table as the referenced or parent table and the other as the dependent or child table. In the parent table, define either a **PRIMARY KEY or UNIQUE** constraint on the shared columns. In the child table, define a **FOREIGN KEY** constraint on the shared columns. The shared columns now comprise a foreign key. Defining additional constraints on the foreign key affects the parent-child relationship

Figure 5-2 Rows That Violate and Satisfy a FOREIGN KEY Constraint

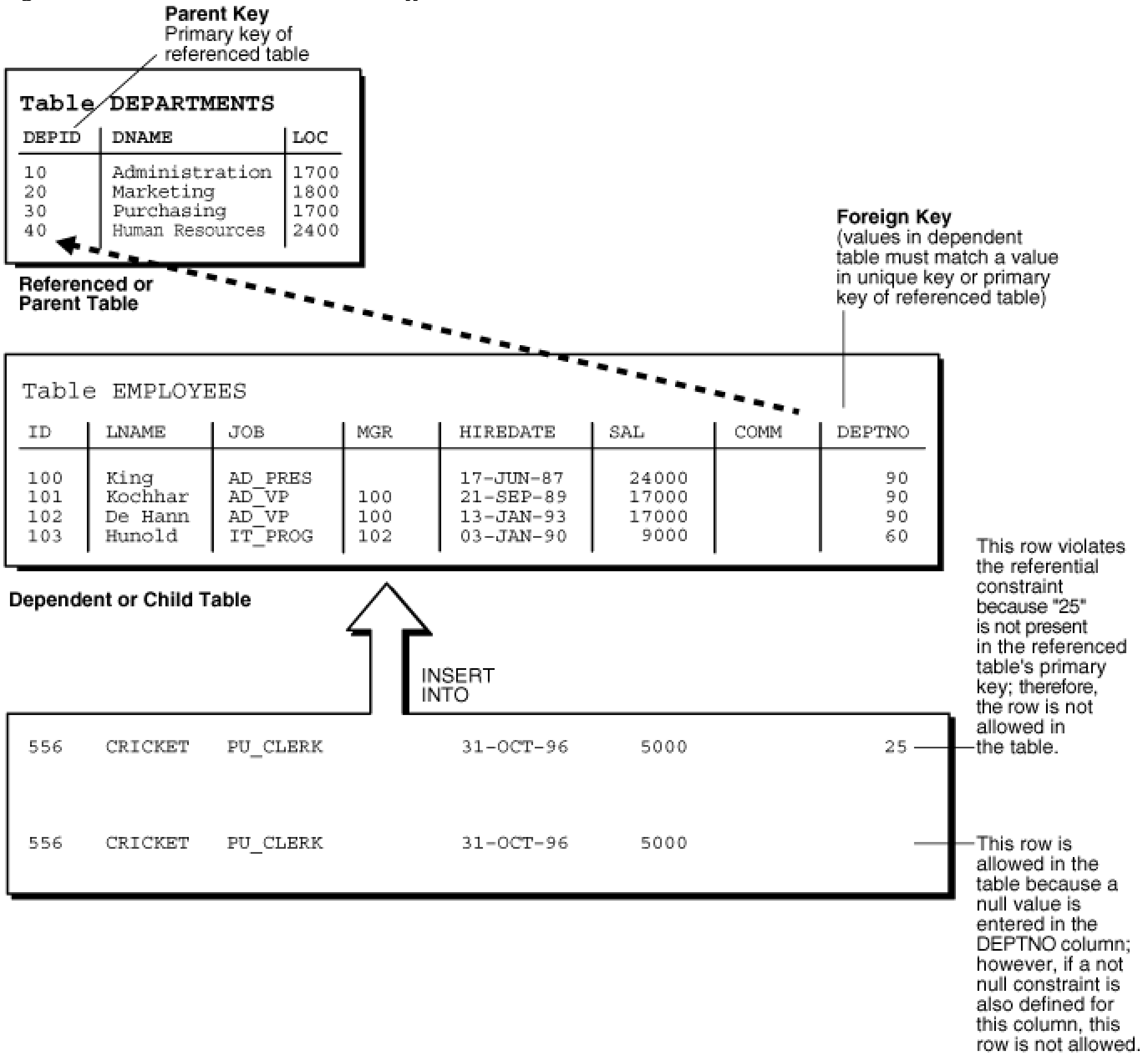

Parent Key
Primary key of referenced table
Referenced or Parent Table
Foreign Key
(values in dependent table must match a value in unique key or primary key of referenced table)
Dependent or Child Table
INSERT INTO
This row violates the referential constraint because "25" is not present in the referenced table's primary key; therefore, the row is not allowed in the table.
This row is allowed in the table because a null value is entered in the DEPTNO column; however, if a not null constraint is also defined for this column, this row is not allowed.

Table DEPARTMENTS

DEPID	DNAME	LOC
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
40	Human Resources	2400

Table EMPLOYEES

ID	LNAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
100	King	AD_PRES		17-JUN-87	24000		90
101	Kochhar	AD_VP	100	21-SEP-89	17000		90
102	De Hann	AD_VP	100	13-JAN-93	17000		90
103	Hunold	IT_PROG	102	03-JAN-90	9000		60

556	CRICKET	PU_CLERK		31-OCT-96	5000		25
556	CRICKET	PU_CLERK		31-OCT-96	5000		

CREATING AND MAINTAING TABLES

The process of creating a table is far more standardized than the CREATE DATABASE statement. Here's the basic syntax for the CREATE TABLE statement:

SYNTAX:

```
CREATE TABLE table_name
(    field1 datatype [ NOT NULL ],
     field2 datatype [ NOT NULL ],
     field3 datatype [ NOT NULL ]...)
```

A simple example of a CREATE TABLE statement follows.

INPUT/OUTPUT:

```
SQL>  CREATE TABLE BILLS (
  2   NAME CHAR(30),
  3   AMOUNT NUMBER,
  4   ACCOUNT_ID NUMBER);

Table created.
```

ANALYSIS:

This statement creates a table named **BILLS.** Within the BILLS table are three fields**: NAME, AMOUNT**, and **ACCOUNT_ID.** The **NAME** field has a data type of character and can store strings up to 30 characters long. The **AMOUNT** and **ACCOUNT_ID** fields can contain number values only.

The following section examines components of the **CREATE TABLE** command.

The Table Name

When creating a table, several constraints apply when naming the table. First, the table name can be no more than 30 characters long. Because Oracle is case insensitive, you can use either uppercase or lowercase for the individual characters. However, the first character of the name must be a letter between A and Z. The remaining characters can be letters or the symbols _, #, $, and @. Of course, the table name must be unique within its schema. The name also cannot be one of the Oracle or SQL reserved words (such as SELECT).

The Field Name

The same constraints that apply to the table name also apply to the field name. However, a field name can be duplicated within the database. The restriction is that the field name must be unique within its table.

The Field's Data Type

If you have ever programmed in any language, you are familiar with the concept of data types, or the type of data that is to be stored in a specific field. For instance, a character data type constitutes a field that stores only character string data

Creating a Table from an Existing Table

The most common way to create a table is with the **CREATE TABLE** command. However, some database management systems provide an alternative method of creating tables, using the format and data of an existing table. This method is useful when you want to select the data out of a table for temporary modification. It can also be useful when you have to create a table similar to the existing table and fill it with similar data.

SYNTAX:
```
CREATE TABLE NEW_TABLE(FIELD1, FIELD2, FIELD3)
AS (SELECT FIELD1, FIELD2, FIELD3
  FROM OLD_TABLE <WHERE...>
```

EXAMPLE:
```
SQL>  CREATE TABLE NEW_BILLS(NAME, AMOUNT, ACCOUNT_ID)
  2   AS (SELECT * FROM BILLS WHERE AMOUNT < 50);

Table created.
```

The ALTER TABLE Statement

The ALTER TABLE statement enables the database administrator or designer to change the structure of a table after it has been created.

The ALTER TABLE command enables you to do two things:

- Add a column to an existing table
- Modify a column that already exists

The syntax for the **ALTER TABLE** statement is as follows:

SYNTAX:
```
ALTER TABLE table_name
 <ADD column_name data_type; |
 MODIFY column_name data_type;>
```

The following command changes the NAME field of the BILLS table to hold 40 characters:

```
SQL>  ALTER TABLE BILLS
  2   MODIFY NAME CHAR(40);

Table altered.


SQL>  ALTER TABLE NEW_BILLS
  2   ADD COMMENTS CHAR(80);
```

```
Table altered.
```

The DROP TABLE Statement

SQL provides a command to completely remove a table from a database. The DROP TABLE command deletes a table along with all its associated views and indexes.

SYNTAX:

```
DROP TABLE table_name;
```

EXAMPLE:

```
SQL> DROP TABLE NEW_BILLS;

Table dropped.
```

INDEXES

Data can be retrieved from a database using two methods. The first method, often called the Sequential Access Method, requires SQL to go through each record looking for a match. This search method is inefficient. Adding indexes to your database enables SQL to use the Direct Access Method. SQL uses a treelike structure to store and retrieve the index's data. Pointers to a group of data are stored at the top of the tree. These groups are called nodes. Each node contains pointers to other nodes. The nodes pointing to the left contain values that are less than its parent node. The pointers to the right point to values greater than the parent node.

The database system starts its search at the top node and simply follows the pointers until it is successful.

SYNTAX:

```
CREATE [UNIQUE | DISTINCT] [CLUSTER] INDEX index_name
ON table_name (column_name [ASC | DESC],
        column_name [ASC | DESC]...)
```

Notice that all of these implementations have several things in common, starting with the basic statement

```
CREATE INDEX index_name ON table_name (column_name, ...)
```

EXAMPLE:

```
SQL> SELECT * FROM BILLS;
NAME                AMOUNT    ACCOUNT_ID
Phone Company            125      1
Power Company            75       1
Record Club            25       2
Software Company          250      1
Cable TV Company          35       3
Joe's Car Palace          350      5
S.C. Student Loan          200      6
Florida Water Company       20       1
U-O-Us Insurance Company    125      5
Debtor's Credit Card       35       4

10 rows selected.

SQL> CREATE INDEX ID_INDEX ON BILLS( ACCOUNT_ID );

Index created.

SQL> SELECT * FROM BILLS;

NAME                AMOUNT    ACCOUNT_ID
Phone Company            125      1
Power Company            75       1
Software Company          250      1
Florida Water Company       20       1
Record Club            25       2
Cable TV Company          35       3
Debtor's Credit Card       35       4
Joe's Car Palace          350      5
U-O-Us Insurance Company    125      5
S.C. Student Loan          200      6

10 rows selected.
```

The BILLS table is sorted by the ACCOUNT_ID field until the index is dropped using the DROP INDEX statement. As usual, the DROP INDEX statement is very straightforward:

SYNTAX:

```
SQL> DROP INDEX index_name;
```

EXAMPLE:

```
SQL> DROP INDEX ID_INDEX;

Index dropped.
```

SEQUENCES

Sequences are database objects from which multiple users can generate unique integers. You can use sequences to automatically generate primary key values.

Creating Sequences

To create a sequence in your schema, you must have the **CREATE SEQUENCE** system privilege. To create a sequence in another user's schema, you must have the **CREATE** ANY **SEQUENCE** privilege.

Create a sequence using the **CREATE SEQUENCE** statement.

```
CREATE SEQUENCE emp_sequence
    INCREMENT BY 1
    START WITH 1
    NOMAXVALUE
    NOCYCLE
    CACHE 10;
```

The **CACHE** option pre-allocates a set of sequence numbers and keeps them in memory so that sequence numbers can be accessed faster.

Altering Sequences

To alter a sequence, your schema must contain the sequence, or you must have the **ALTER ANY SEQUENCE** system privilege. You can alter a sequence to change any of the parameters that define how it generates sequence numbers except the sequence's starting number. To change the starting point of a sequence, drop the sequence and then re-create it.

Alter a sequence using the **ALTER SEQUENCE** statement. For example, the following statement alters the **emp_sequence**:

```
ALTER SEQUENCE emp_sequence
   INCREMENT BY 10
   MAXVALUE 10000
   CYCLE
   CACHE 20;
```

Dropping Sequences

You can drop any sequence in your schema. To drop a sequence in another schema, you must have the **DROP ANY SEQUENCE** system privilege. If a sequence is no longer required, you can drop the sequence using the **DROP SEQUENCE** statement. For example, the following statement drops the **order_seq** sequence:

```
DROP SEQUENCE order_seq;
```

VIEWS

A view is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called base tables. Base tables might in turn be actual tables or might be views themselves. All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.

Creating Views

To create a view, you must meet the following requirements:

To create a view in your schema, you must have the **CREATE VIEW** privilege

The following statement creates a view on a subset of data in the **emp** table:

```
CREATE VIEW sales_staff AS SELECT empno, ename, deptno FROM emp WHERE deptno = 10
```

Join Views

You can also create views that specify more than one base table or view in the **FROM** clause. These are called join views. The following statement creates the **division1_staff** view that joins data from the **emp** and **dept** tables:

```
CREATE VIEW division1_staff AS SELECT ename, empno, job, dname FROM emp, dept

   WHERE emp.deptno IN (10, 30) AND emp.deptno = dept.deptno;
```

USERS, PRIVILEGES AND ROLES

Oracle relies on a mechanism that allows you to register a person, called user. Each registered user has an access password , which must be provided in various situations. Each user is then assigned individual privileges or roles.

The types of users and their roles and responsibilities depend on the database site. A small site can have one database administrator who administers the database for application developers and users

SYNTAX:

```
CREATE USER username
  IDENTIFIED BY password;
```

EXAMPLE:

CREATE USER john IDENTIFIED BY abcd1234;

User JOHN created.

Let's use the **john** account to log in the database.
Launch the SQL*Plus program and enter the following information:

Enter user-name: john
Enter password:<john_password>
Oracle issued the following error:

ERROR: ORA-01045:
user JOHN lacks CREATE SESSION privilege; logon denied

To enable the user **john** to log in, you need to grant the **CREATE SESSION** system privilege to the user **john** by using the following statement:

GRANT CREATE SESSION TO john;
Now, the user john should be able to log in the database.
Enter user-name: john
Enter password:

Connected to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

PRIVILEGES

A user privilege is a right to execute a particular type of SQL statement, or a right to access another user's object. The types of privileges are defined by Oracle.

System Privileges

There are over 100 distinct system privileges. Each system privilege allows a user to perform a particular database operation or class of database operations.

Object Privileges

Each type of object has different privileges associated with it.

You can specify ALL [PRIVILEGES] to grant or revoke all available object privileges for an object. ALL is not a privilege; rather, it is a shortcut, or a way of granting or revoking all **object** privileges with one word in GRANT and REVOKE statements

USER ROLES

A **role** groups several privileges and roles, so that they can be granted to and revoked from users simultaneously. A role must be enabled for a user before it can be used by the user.

Oracle provides some predefined roles to help in database administration. These roles, listed in Table 25-1, are automatically defined for Oracle databases when you run the standard scripts that are part of database creation. You can grant privileges and roles to, and revoke privileges and roles from, these predefined roles in the same way as you do with any role you define.

Granting System Privileges and Roles

You can grant system privileges and roles to other users and roles using the GRANT statement. The following privileges are required:

- To grant a system privilege, you must have been granted the system privilege with the ADMIN **OPTION** or have been granted the **GRANT ANY PRIVILEGE** system privilege.
- To grant a role, you must have been granted the role with the ADMIN OPTION or have been granted the **GRANT ANY ROLE** system privilege.

The following statement grants the system privilege **CREATE SESSION** and the **accts_pay** role to the user **jward**:

```
GRANT CREATE SESSION, accts_pay TO jward;
```

The following statement grants the **SELECT, INSERT**, and **DELETE** object privileges for all columns of the **emp** table to the users **jfee** and **tsmith**:

```
GRANT SELECT, INSERT, DELETE ON emp TO jfee, tsmith;
```

To grant all object privileges on the **salary** view to the user jfee, use the ALL keyword, as shown in the following example:

```
GRANT ALL ON salary TO jfee;
```

Revoking System Privileges and Roles

You can revoke system privileges and roles using the SQL statement **REVOKE.**

Any user with the **ADMIN OPTION** for a system privilege or role can revoke the privilege or role from any other database user or role. The revoker does not have to be the user that originally granted the privilege or role. Users with GRANT ANY ROLE can revoke *any* role.

The following statement revokes the **CREATE TABLE** system privilege and the **accts_rec** role from **tsmith**:

```
REVOKE CREATE TABLE, accts_rec FROM tsmith;
```

Revoking Object Privileges

The REVOKE statement is used to revoke object privileges. To revoke an object privilege, you must fulfill one of the following conditions:

- You previously granted the object privilege to the user or role.
- You possess the **GRANT ANY OBJECT PRIVILEGE** system privilege that enables you to grant and revoke privileges on behalf of the object owner.

You can only revoke the privileges that you, the grantor, directly authorized, not the grants made by other users to whom you granted the GRANT OPTION. However, there is a cascading effect. The object privilege grants propagated using the GRANT OPTION are revoked if a grantor's object privilege is revoked.

Assuming you are the original grantor, the following statement revokes the **SELECT** and **INSERT** privileges on the **emp** table from the users **jfee** and **tsmith:**

```
REVOKE SELECT, insert ON emp FROM jfee, tsmith;
```

The following statement revokes all object privileges for the **dept** table that you originally granted to the human_**resource** role

```
REVOKE ALL ON dept FROM human_resources;
```

SYNONYMS

A synonym is an alias for a schema object. Synonyms can provide a level of security by masking the name and owner of an object and by providing location transparency for remote objects of a distributed database. Also, they are convenient to use and reduce the complexity of SQL statements for database users.

Synonyms allow underlying objects to be renamed or moved. You can create both public and private synonyms.

A **public** synonym is owned by the special user group named PUBLIC and is accessible to every user in a database. A **private** synonym is contained in the schema of a specific user and available only to the user and the user's grantees.

Creating Synonyms

To create a private synonym in your own schema, you must have the **CREATE SYNONYM** privilege. To create a private synonym in another user's schema, you must have the **CREATE ANY SYNONYM** privilege. To create a public synonym, you must have the **CREATE PUBLIC SYNONYM** system privilege.

Create a synonym using the **CREATE SYNONYM** statement. The underlying schema object need not exist, nor do you need privileges to access the object. The following statement creates a public synonym named **public_emp** on the **emp** table contained in the schema of **jward:**

```
CREATE PUBLIC SYNONYM public_emp FOR jward.emp;
```

Dropping Synonyms

You can drop any private synonym in your own schema. To drop a private synonym in another user's schema, you must have the **DROP ANY SYNONYM** system privilege. To drop a public synonym, you must have the **DROP PUBLIC SYNONYM** system privilege.

Drop a synonym that is no longer required using **DROP SYNONYM** statement. To drop a private synonym, omit the **PUBLIC** keyword. To drop a public synonym, include the **PUBLIC** keyword.

For example, the following statement drops the private synonym named emp:

```
DROP SYNONYM emp;
```

UNIT V

1. PL/SQL
2. TRIGGERS
3. STORED PROCEDURE AND FUNCTIONS
4. PACKAGE
5. CURSORS
6. TRANSACTIONS

PL/SQL

PL/SQL, the Oracle procedural extension of SQL, is a completely portable, high-performance transaction-processing language.

Main Features of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.

When a problem can be solved using SQL, you can issue SQL statements from your PL/SQL programs, without learning new APIs.

Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap run-time errors.

You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

PL/SQL Blocks

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements.

A PL/SQL block is defined by the keywords `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required.

Declarations are local to the block and cease to exist when the block completes execution, helping to avoid cluttered namespaces for variables and subprograms.

Example 1-1 PL/SQL Block Structure

```
DECLARE    -- Declarative part (optional)

  -- Declarations of local types, variables, & subprograms

BEGIN      -- Executable part (required)

  -- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)

  -- Exception handlers for exceptions raised in executable part]



END;
```

Declaring PL/SQL Variables

A PL/SQL variable can have any SQL data type (such as `CHAR`, `DATE`, or `NUMBER`) or a PL/SQL-only data type (such as `BOOLEAN` or `PLS_INTEGER`).

Example 1-2 declares several PL/SQL variables. One has a PL/SQL-only data type; the others have SQL data types.

Example 1-2 PL/SQL Variable Declarations

```
       part_number       NUMBER(6);     -- SQL data type

     part_name         VARCHAR2(20);  -- SQL data type

     in_stock          BOOLEAN;       -- PL/SQL-only data type

     part_price        NUMBER(6,2);   -- SQL data type
```

Example 1-3 Assigning Values to Variables with the Assignment Operator

```
    DECLARE  -- You can assign values here

    wages          NUMBER;

      country        VARCHAR2(128);

   counter        NUMBER := 0;

   done           BOOLEAN;

   BEGIN  -- You can assign values here too

     wages := (hours_worked * hourly_salary) + bonus;

     country := 'France';

     country := UPPER('Canada');

    done := (counter > 100);

  END;

  /
```

%TYPE Attribute

The `%TYPE` attribute provides the data type of a variable or database column. This is particularly useful when declaring variables that will hold database values. For example, assume there is a column named `last_name` in a table named `employees`. To declare a variable named `v_last_name` that has the same data type as column `last_name`, use dot notation and the `%TYPE` attribute, as follows:

```
v_last_name employees.last_name%TYPE;
```

%ROWTYPE Attribute

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The `%ROWTYPE` attribute provides a record type that represents a row in a table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. See Cursors.

Columns in a row and corresponding fields in a record have the same names and data types. In the following example, you declare a record named `dept_rec`, whose fields have the same names and data types as the columns in the `departments` table:

```
dept_rec departments%ROWTYPE; -- declare record variable
```

You use dot notation to reference fields, as follows:

```
v_deptid := dept_rec.department_id;
```

PL/SQL Control Structures

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate database data, it lets you process the data using flow-of-control statements.

Conditional Control

Often, it is necessary to take alternative actions depending on circumstances. The `IF-THEN-ELSE` statement lets you execute a sequence of statements conditionally. The `IF` clause checks a condition, the `THEN` clause defines what to do if the condition is true and the `ELSE` clause defines what to do if the condition is false or nul

Iterative Control

`LOOP` statements let you execute a sequence of statements multiple times. You place the keyword `LOOP` before the first statement in the sequence and the keywords `END` `LOOP` after the last statement in the sequence. The following example shows the simplest kind of loop, which repeats a sequence of statements continually:

```
LOOP

  -- sequence of statements

END LOOP;
```

TRIGGERS

A trigger is a named program unit that is stored in the database and **fired** (executed) in response to a specified event. The specified **event** is associated with either a table, a view, a schema, or the database, and it is one of the following:

- A database manipulation (DML) statement (`DELETE`, `INSERT`, or `UPDATE`)

- A database definition (DDL) statement (`CREATE`, `ALTER`, or `DROP`)
- A database operation (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`)

The trigger is said to be **defined on** the table, view, schema, or database.

Trigger Types

A **DML trigger** is fired by a DML statement, a **DDL trigger** is fired by a DDL statement, a **`DELETE` trigger** is fired by a `DELETE` statement, and so on.

An **`INSTEAD OF` trigger** is a DML trigger that is defined on a view (not a table). The database fires the `INSTEAD OF` trigger instead of executing the triggering DML statement. For more information, see Modifying Complex Views (INSTEAD OF Triggers).

A **system trigger** is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

A **simple trigger** can fire at exactly one of the following **timing points**:

- Before the triggering statement executes
- After the triggering statement executes
- Before each row that the triggering statement affects
- After each row that the triggering statement affects

A **compound trigger** can fire at more than one timing point. Compound triggers make it easier to program an approach where you want the actions you implement for the various timing points to share common data. For more information, see Compound Triggers.

Trigger States

A trigger can be in either of two states:

Enabled. An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

Disabled. A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

Uses of Triggers

Triggers supplement the standard capabilities of your database to provide a highly customized database management system. For example, you can use triggers to:

- Automatically generate derived column values
- Enforce referential integrity across nodes in a distributed database
- Enforce complex business rules
- Provide transparent event logging
- Provide auditing

// TRIGGER PROGRAM

STORED PROCEDURE AND FUNCTION

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

CREATE PROCEDURE *procedure_name*
AS
pl/sql_statement block;

Execute a Stored Procedure

EXEC *procedure_name*;

//Example program

FUNCTION

A function is same as procedure ,but it returns a value.

The `CREATE FUNCTION` statement creates or replaces a standalone stored function or a call specification.

A **standalone stored function** is a function (a subprogram that returns a single value) that is stored in the database.

Creating a Function: Examples The following statement creates the function `get_bal` on the sample table `oe.orders`:

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)

   RETURN NUMBER

   IS acc_bal NUMBER(11,2);

   BEGIN

      SELECT order_total

      INTO acc_bal

      FROM orders
      WHERE customer_id = acc_no;

      RETURN(acc_bal);

    END;

/
```

The get_bal function returns the balance of a specified account.

When you call the function, you must specify the argument acc_no, the number of the account whose balance is sought. The data type of acc_no is NUMBER.

The function returns the account balance. The RETURN clause of the CREATE FUNCTION statement specifies the data type of the return value to be NUMBER.

The function uses a SELECT statement to select the balance column from the row identified by the argument acc_no in the orders table. The function uses a RETURN statement to return this value to the environment in which the function is called.

The function created in the preceding example can be used in a SQL statement. For example:

```
SELECT get_bal(165) FROM DUAL;



GET_BAL(165)

------------

        2519
```

PL/SQL PACKAGE

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.

You can think of the spec as an interface and of the body as a black box. You can debug, enhance, or replace a package body without changing the package spec.

To create a package spec, use the CREATE PACKAGE Statement. To create a package body, use the CREATE PACKAGE BODY Statement.

The spec holds public declarations, which are visible to stored subprograms and other code outside the package. You must declare subprograms at the end of the spec after all other items (except pragmas that name a specific function; such pragmas must follow the function spec).

The body holds implementation details and private declarations, which are hidden from code outside the package. Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps.

PL/SQL Package Specification

The package specification contains public declarations. The declared items are accessible from anywhere in the package and to any other subprograms in the same schema

PL/SQL Package Body

The package body contains the implementation of every cursor and subprogram declared in the package spec. Subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec. If a subprogram spec is not included in the package spec, that subprogram can only be invoked by other subprograms in the same package. A package body must be in the same schema as the package spec.

Example 10-2 Matching Package Specifications and Bodies

```
CREATE PACKAGE emp_bonus AS

   PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);

END emp_bonus;

/

CREATE PACKAGE BODY emp_bonus AS

-- the following parameter declaration raises an exception

-- because 'DATE' does not match employees.hire_date%TYPE

-- PROCEDURE calc_bonus (date_hired DATE) IS

-- the following is correct because there is an exact match

   PROCEDURE calc_bonus

     (date_hired employees.hire_date%TYPE) IS

   BEGIN

     DBMS_OUTPUT.PUT_LINE

       ('Employees hired on ' || date_hired || ' get bonus.');

   END;

END emp_bonus;

/
```

DBMS_OUTPUT Package

DBMS_OUTPUT package enables you to display output from PL/SQL blocks, subprograms, packages, and triggers. The package is especially useful for displaying PL/SQL debugging information. The procedure PUT_LINE outputs information to a buffer that can be read by another trigger, subprogram, or package. You display the information by invoking the procedure GET_LINE or by setting SERVEROUTPUT ON in SQL*Plus. Example 10-4 shows how to display output from a PL/SQL block.

Example 10-4 Using PUT_LINE in the DBMS_OUTPUT Package

```
REM set server output to ON to display output from DBMS_OUTPUT

SET SERVEROUTPUT ON

BEGIN

  DBMS_OUTPUT.PUT_LINE

    ('These are the tables that ' || USER || ' owns:');

  FOR item IN (SELECT table_name FROM user_tables)

    LOOP

      DBMS_OUTPUT.PUT_LINE(item.table_name);

    END LOOP;

END;

/
```

CURSORS

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in 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

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE state rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatical its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, SELECT INTO statement.

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory

- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

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

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

TRANSACTIONS

Transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a `COMMIT` or `ROLLBACK` statement or implicitly when a DDL statement is issued.

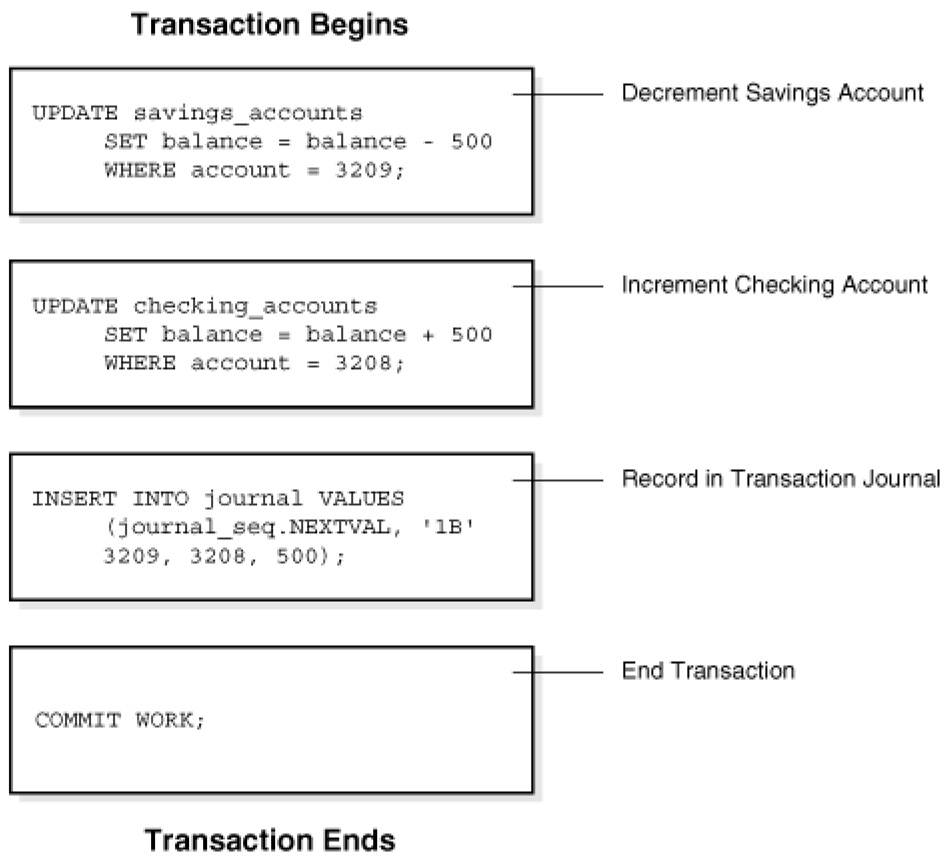
To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle Database must allow for two situations. If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back so that the balance of all accounts is correct.

Figure 4-1 illustrates the banking transaction example.

Figure 4-1 A Banking Transaction


Transaction Begins
Decrement Savings Account
Increment Checking Account
Record in Transaction Journal
End Transaction
Transaction Ends

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

```
COMMIT WORK;
```

Commit Transactions

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that modifies data is committed, the following has occurred:

- Oracle Database has generated undo information. The undo information contains the old data values changed by the SQL statements of the transaction.
- Oracle Database has generated redo log entries in the redo log buffer of the SGA. The redo log record contains the change to the data block and the change to the rollback block. These changes may go to disk before a transaction is committed.
- The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction is committed.

Rollback of Transactions

Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction. Oracle Database uses undo tablespaces (or rollback segments) to store old values. The redo log contains a record of changes.

Oracle Database lets you roll back an entire uncommitted transaction. Alternatively, you can roll back the trailing portion of an uncommitted transaction to a marker called a savepoint.

All types of rollbacks use the same procedures:

- Statement-level rollback (due to statement or deadlock execution error)
- Rollback to a savepoint
- Rollback of a transaction due to user request
- Rollback of a transaction due to abnormal process termination
- Rollback of all outstanding transactions when an instance terminates abnormally
- Rollback of incomplete transactions during recovery

The Two-Phase Commit Mechanism

In a distributed database, Oracle Database must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

A **distributed transaction** is a transaction that includes one or more statements that update data on two or more distinct nodes of a distributed database.

A **two-phase commit** mechanism guarantees that *all* database servers participating in a distributed transaction either all commit or all undo the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

The Oracle Database two-phase commit mechanism is completely transparent to users who issue distributed transactions