Department of Computer Science

Aligarh Muslim University

Aligarh

COURSE CSM- 5541: ADVANCE DBMS AND DBA

Prof. Mohammad Ubaidullah Bokhari

Unit 1 – Part B.

- Object Oriented Database
- Object Relational Databases
- Temporal Database concept
- Distributed Database Management: Reference Architecture, levels of distribution transparency, Distributed database design, Distributed Query Processing and Optimization, Distributed Transaction Modeling.

Object Oriented Database

An object-oriented database is a database that subscribes to a model with information represented by objects. Object-oriented databases are a niche offering in the relational database management system (RDBMS) field and are not as successful or well-known as mainstream database engines.

As the name implies, the main feature of object-oriented databases is allowing the definition of objects, which are different from normal database objects. Objects, in an object-oriented database, reference the ability to develop a product, then define and name it. The object can then be referenced, or called later, as a unit without having to go into its complexities. This is very similar to objects used in object-oriented programming.

A real-life parallel to objects is a car engine. It is composed of several parts: the main cylinder block, the exhaust system, intake manifold and so on. Each of these is a standalone component; but when machined and bolted into one object, they are now collectively referred to as an engine. Similarly, when programming one can define several components, such as a vertical line intersecting a perpendicular horizontal line while both lines have a graded measurement. This object can then be collectively labeled a graph. When utilizing the ability to plot components, there is no need to first define a graph; but rather the instance of the created graph can be called.

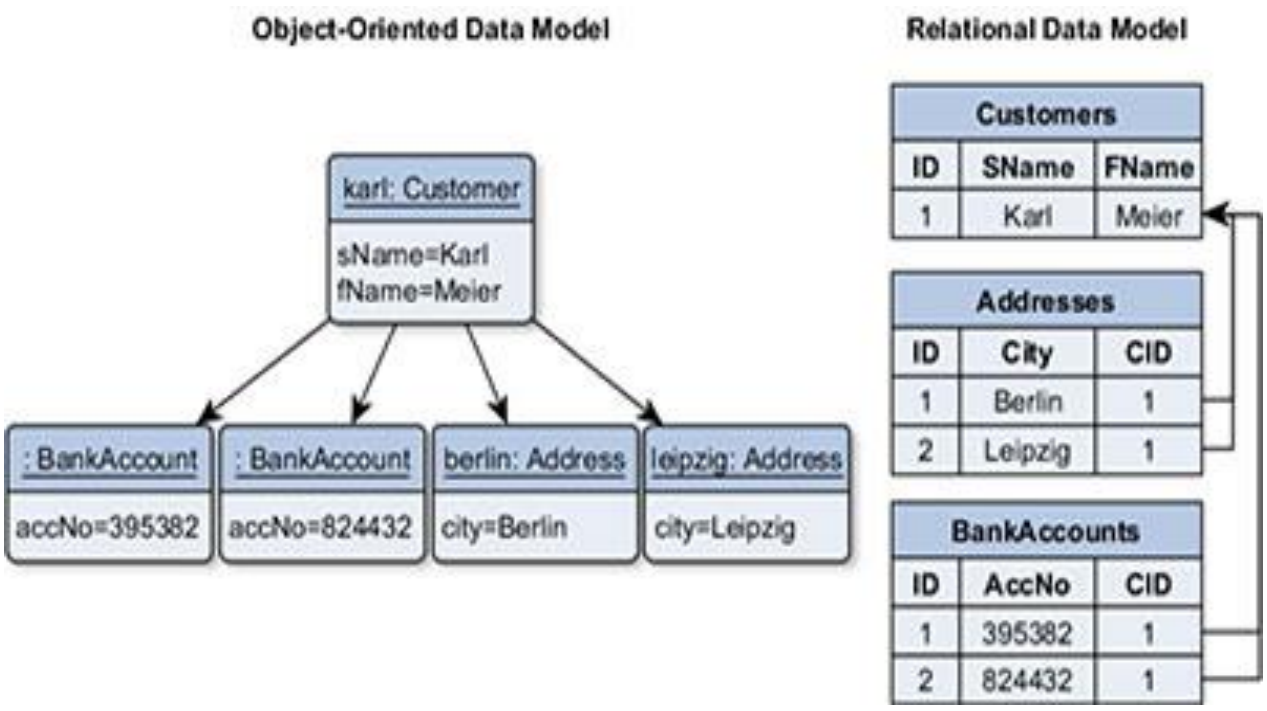
Examples of object-oriented database engines include db4o, Smalltalk and Cache.

KNOW THE DIFFERENCE-LEARN HOW OBJECT ORIENTED DATABASE IS DIFFERENT FROM RDBMS

Before moving on to maters of finding out how something works, we first need to indulge our curiosity and learn the purpose of the analyzed subject. Being that the topic here is database management, the first step is to learn what they are exactly. Database management or DMBS is an application that interacts with users, the database itself, and other applications to monitor, capture or analyze the data. It's being used for personal, social or business purposes to share ideas and information in general. Now that we've covered the basics let's find out about the difference in database management.

The main difference between these two systems of database managing is that the way they access and process information. In relational database managing system data is transferred in a relational way. This means that each access control table that stores data has a key field that identifies a row. In both network and hierarchical database accessing information is performed differently. Relational database connects data tables with rows to transfer information.

In object oriented database management we have an entirely different approach where the information is represented in objects. If you are familiar with object-oriented programming, you'll recognize the pattern. The main difference between object oriented database management system and a relation model is their approach on a digital transcript of information and the programming language. When data is stored in an object-oriented database system, it is in the form of an object. Each object consists of two elements where one is a piece of data (sound, text, video, etc.) and another is instruction for software. The instruction determines how the information will be transferred to another data file and the piece of data regulated where a specific type of this information will be heading to. In this complex system of managing data, it's not enough to simply know a specific language but to understand commands as well.



Definition and Overview of ODBMS

The **ODBMS** which is an abbreviation for **object oriented database management system**, is the data model in which data is stored in form of objects, which are instances of classes. These classes and objects together makes an object oriented data model.

Components of Object Oriented Data Model:

The OODBMS is based on three major components, namely: Object structure, Object classes, and Object identity. These are explained as following below.

1. Object Structure:
The structure of an object refers to the properties that an object is made up of. These properties of an object are referred to as an attribute. Thus, an object is a real world, world entity with certain attributes that makes up the object structure. Also an object encapsulates the data code into a single unit which in turn provides data abstraction by hiding the implementation details from the user.
The object structure is further composed of three types of components: Messages, Methods, and Variables. These are explained as following below.

1. **Messages –**
   A message provides an interface or acts as a communication medium between an object and the outside world. A message can be of two types:
   - **Read-only message:** If the invoked method does not change the value of a variable, then the invoking message is said to be a read-only message.
   - **Update message:** If the invoked method changes the value of a variable, then the invoking message is said to be an update message.
2. **Methods –**
   When a message is passed then the body of code that is executed is known as a method. Every time when a method is executed, it returns a value as output. A method can be of two types:
   - **Read-only method:** When the value of a variable is not affected by a method, then it is known as read-only method.
   - **Update-method:** When the value of a variable changes by a method, then it is known as an update method.
3. **Variables –**
   It stores the data of an object. The data stored in the variables makes the object distinguishable from one another.

2. Object Classes:
An object which is a real world entity is an instance of a class. Hence first we need to define a class and then the objects are made which differ in the values they store but share the same class definition. The objects in turn corresponds to various messages and variables stored in it.

An OODBMS also supports inheritance in an extensive manner as in a database there may be many classes with similar methods, variables and messages. Thus, the concept of class hierarchy is maintained to depict the similarities among various classes.

The concept of encapsulation that is the data or information hiding is also supported by object oriented data model. And this data model also provides the facility of abstract data types apart from the built-in data types like char, int, float. ADT's are the user defined data types that hold the values within it and can also have methods attached to it.

Thus, OODBMS provides numerous facilities to its users, both built-in and user defined. It incorporates the properties of an object oriented data model with a database management system, and supports the concept of programming paradigms like classes and objects along with the support for other concepts like encapsulation, inheritance and the user defined ADT's (abstract data types).

When the database techniques are combined with object oriented concepts, the result is an object oriented management system (ODBMS). Today's trend in programming languages is to utilize objects, thereby making OODBMS is ideal for Object Oriented programmers because they can develop the product, store them as objects, and can replicate or modify existing objects to make new objects within the OODBMS. Object databases based on persistent programming acquired a niche in application areas such as engineering and spatial databases, telecommunications, and scientific areas such as high energy physics and molecular biology.

List of Object Oriented Database Standards

some of the Object oriented DBMS standards are

- Object data management group
- Object database standard ODM 6.2.0
- Object query language
- Object query language support of SQL 92

Object Oriented Data Model(OODM)

Object oriented data models are a logical data models that capture the semantics of objects supported on object oriented programming. OODMs implement conceptual models directly and can represent complexities that are beyond the capabilities of relational systems. OODBs have adopted many of the concepts that were developed originally for object oriented programming language. An object oriented database is a collection of objects defined by an object oriented data model. An object oriented database can extend the existence of objects so that they are stored permanently. Therefore, the objects persist beyond program termination and can be retrieved later and shared by other programs.

C**haracteristics of Object oriented database**

The characteristics of object oriented database are listed below.

- It keeps up a direct relation between real world and database objects as if objects do not loose their integrity and identity.
- OODBs provide system generated object identifier for each object so that an object can easily be identified and operated upon.
- OODBs are extensible, which identifies new data types and the operations to be performed on them.
- Provides encapsulation, feature which, the data representation and the methods implementation are hidden from external entities.
- Also provides inheritance properties in which an object inherits the properties of other objects.

Object, Attributes and Identity

- *Attributes :* The attributes are the characteristics used to describe objects. Attributes are also known as instance variables. When attributes are assigned values at a given time, it is assumed that the object is in a given state at that time.
- *Object :* An object is an abstract representation of the real world entity which has a unique identity, embedded properties, and the ability to interact with other objects and itself.
- *Identity :* The identity is an external identifier- the object ID- maintained for each object. The Object ID is assigned by the system when the object is created, and cannot be changed. It is unlike the relational database, for example, where a data value stored within the object is used to identify the object.

Object oriented methodologies

There are certain object oriented methodologies are use in OODB. These are:

- *Class:* A class is assumed as a group of objects with the same or similar attributes and behavior.
- *Encapsulation:* It is the property that the attributes and methods of an object are hidden from outside world. A published interface is used to access an object's methods.
- *Inheritance:* It is the property which, when classes are arranged in a hierarchy, each class assumes the attributes and methods of its ancestors. For example, class students are the ancestor of undergraduate students and post graduate students.
- *Polymorphism :* It allows several objects to represent to the same message in different ways. In the object oriented database model, complex objects are modeled more naturally and easily.

Benefit of object orientation in programming language

The benefits of object orientation in programming language are:

- Minimizes number of lines of code
- Reduces development time
- Increases the reusability of codes
- Makes code maintenance easier
- Increased productivity of programmers

Merits of Object oriented database

OODBs provide the following merits

- OODBs allow for the storage of complex data structures that cannot be easily stored using conventional database terminology.
- OODBs support all the persistence required for object oriented applications.
- OODBs contain active object servers which support both distribution of data and distribution of work.

Advantages of Object oriented data model over Relational model

When compared with the relational model, the object oriented data model has the following advantages.

- *Reusability:* generic objects can be defined and then reused in numerous application.
- *Complex data types: C*an manage complex data such as document, graphics, images, voice messages, etc.
- *Distributed databases:* Due to mode of communication between objects, OODBMS can support distribution of data across networks more easily.

Object oriented model vs Entity Relationship model

An entity is simply a collection of variables or data items.

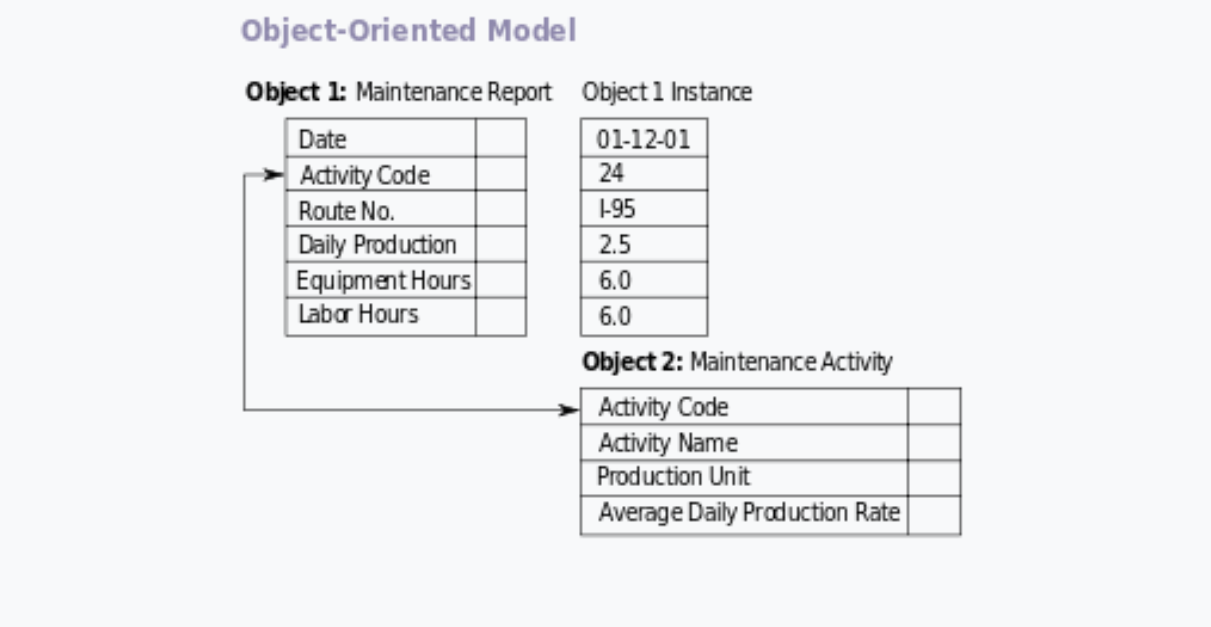
When an application is running, the ER Model will tells how the application's data will be persisted but the Object Oriented Model will decide that how that data will be stored in the memory.

Advantages of OODB over RDBMS

Object oriented database advantages over RDBMS:

- Objects do not require assembly and dis-assembly saving coding time and execution time to assemble or disassemble objects.
- Reduced paging
- Easier navigation
- Better concurrency control
- Data model is based on the real world
- Works well for distributed architectures
- Less code required when applications are object oriented.

An **object-relational database** (**ORD**), or **object-relational database management system** (**ORDBMS**), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data types and methods.



An object-relational database can be said to provide a middle ground between relational databases and object-oriented databases. In object-relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

ORD is said to be the middleman between relational and object-oriented databases because it contains aspects and characteristics from both models. In ORD, the basic approach is based on RDB, since the data is stored in a traditional database and manipulated and accessed using queries written in a query language like SQL. However, ORD also showcases an object-oriented characteristic in that the database is considered an object store, usually for software that is written in an object-oriented programming language. Here, APIs are used to store and access the data as objects.

One of ORD's aims is to bridge the gap between conceptual data modeling techniques for relational and object-oriented databases like the entity-relationship diagram (ERD) and object-relational mapping (ORM). It also aims to connect the divide between relational databases and the object-oriented modeling techniques that are usually used in programming languages like Java, C# and C++.

Traditional RDBMS products concentrate on the efficient organization of data that is derived from a limited set of data-types. On the other hand, an ORDBMS has a feature that allows developers to build and innovate their own data types and methods, which can be applied to the DBMS. With this, ORDBMS intends to allow developers to increase the abstraction with which they view the problem area.

An object-relational database supports the following data types and extensibility:

- Alphanumeric data (such as character strings, integers, decimal, floating point, and date)
- Simple large objects (TEXT and BYTE data types)
- Smart large objects (BLOB and CLOB data types)
- User-defined types (opaque and distinct types)
- Complex data types (composites of existing data types)
- User-defined routines
- Operator functions
- User-defined casts
- User-defined aggregates
- Type and table inheritance
- DataBlade® modules
- User-defined virtual processors
- User-defined access methods

RDBMS Vs ORDB

An RDBMS might commonly involve SQL statements such as these:

```
CREATE TABLE Customers  (
    Id          CHAR(12)     NOT NULL PRIMARY KEY,
    Surname     VARCHAR(32) NOT NULL,
    FirstName   VARCHAR(32) NOT NULL,
    DOB         DATE         NOT NULL
 );
 SELECT InitCap(Surname) || ', ' || InitCap(FirstName)
   FROM Customers
  WHERE Month(DOB) = Month(getdate())
    AND Day(DOB) = Day(getdate())
```

Most current SQL databases allow the crafting of custom functions, which would allow the query to appear as:

```
SELECT Formal(Id)
  FROM Customers
 WHERE Birthday(DOB) = Today()
```

In an object-relational database, one might see something like this, with user-defined data-types and expressions such as `BirthDay()`:

```
CREATE TABLE Customers (
  Id            Cust_Id     NOT NULL  PRIMARY KEY,
  Name          PersonName  NOT NULL,
  DOB           DATE        NOT NULL
);
SELECT Formal( C.Id )
  FROM Customers C
 WHERE BirthDay ( C.DOB ) = TODAY;
```

The object-relational model can offer another advantage in that the database can make use of the relationships between data to easily collect related records. In an address bookapplication, an additional table would be added to the ones above to hold zero or more addresses for each customer. Using a traditional RDBMS, collecting information for both the user and their address requires a "join":

```
SELECT InitCap(C.Surname) || ', ' || InitCap(C.FirstName), A.city
  FROM Customers C join Addresses A ON A.Cust_Id=C.Id -- the join
 WHERE A.city="New York"
```

The same query in an object-relational database appears more simply:

```
SELECT Formal( C.Name )
  FROM Customers C
 WHERE C.address.city="New York" -- the linkage is 'understood' by the
ORDB
```

Temporal Database concept

Definition - What does *Temporal Database* mean?

A temporal database is a database that has certain features that support time-sensitive status for entries. Where some databases are considered current databases and only support factual data considered valid at the time of use, a temporal database can establish at what times certain entries are accurate.

Dating from the early 1990s, development communities looked to develop specific guidelines for temporal databases in order to represent time frames for entries. Elements of temporal databases include "valid time" indicators and "transaction time" indicators. Experts describe the "valid time" as the time an entry is expected to be true or valid, and "transaction time" as an internal reference for databases. The valid time tables are also called "application time" tables, while transaction time tables can be referred to as "system version" tables.

Technologies including Oracle, Teradata and SQL have versions with temporal feature support.

Different uses of temporal databases require radically different types of development. For example, in a database of customer, patient or citizen data, indicators for individual people will follow a kind of life cycle timeline that can be created according to time frames for comment life events. By contrast, many industrial processes using temporal databases need extremely short valid time and transaction time indicators. These are rigidly implemented depending on length of time for various parts of business processes.

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been

```
T1: CREATE TRIGGER Total_sal1
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD ROW AS O, NEW ROW AS N
    FOR EACH ROW
    WHEN  ( N.Dno IS NOT NULL )
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + N.salary – O.salary
    WHERE Dno = N.Dno;

T2: CREATE TRIGGER Total_sal2
    AFTER UPDATE OF Salary ON EMPLOYEE
    REFERENCING OLD TABLE AS O, NEW TABLE AS N
    FOR EACH STATEMENT
    WHEN  EXISTS ( SELECT *FROM N WHERE N.Dno IS NOT NULL ) OR
          EXISTS ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
    UPDATE DEPARTMENT AS D
    SET D.Total_sal = D.Total_sal
    + ( SELECT SUM (N.Salary) FROM N WHERE D.Dno=N.Dno )
    – ( SELECT SUM (O.Salary) FROM O WHERE D.Dno=O.Dno )
    WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );
```

Developed since the early days of database usage. However, in creating these applications, it is mainly left to the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include *healthcare,* where patient histories need to be maintained; *insurance,* where claims and accident histories are required as well as information about the times when insurance policies are in effect; *reservation systems* in general (hotel, airline, car rental, train, and so on), where information on the dates and times when reservations are in effect are required; *scientific databases,* where data collected from experiments includes the time when each data is measured; and so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee. In the UNIVERSITY data-base, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE, the grade history of a STUDENT, and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. However, users often attempt to simplify or ignore temporal aspects because of the complexity that they add to their applications.

Introduction to some of the concepts that have been developed to deal with the complexity of temporal database applications need to be discussed. Section 1 gives an overview of how time is represented in databases, the different types of temporal information, and some of the different dimensions of time that may be needed. Section 2 discusses how time can be incorporated into relational databases. Section 3 gives some additional options for representing time that are possible in database models that allow complex-structured objects, such as object databases. Section 4 introduces operations for querying temporal databases, and gives a brief overview of the TSQL2 language, which extends SQL with temporal concepts. Section 5 focuses on time series data, which is a type of temporal data that is very important in practice.

1. Time Representation, Calendars, and Time Dimensions

For temporal databases, time is considered to be an *ordered sequence* of **points** in some **granularity** that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second using this granularity. In reality, each second is a (short) *time duration,* not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchers have used the term **chronon** instead of point to describe this minimal granularity for a particular application. The main consequence of choosing a minimum granularity—say, one second—is that events occurring within the same second will be considered to be*simultaneous events,* even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (western), Chinese, Islamic, Hindu, Jewish, Coptic, and so on) with different reference points. A **calendar** organizes time into different time units for convenience. Most calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth’s rotation around its axis), and 7 days into a week. Further grouping of

days into months and months into years either follow solar or lunar natural phenomena, and are generally irregular. In the Gregorian calendar, which is used in most western countries, days are grouped into months that are 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.

In SQL2, the temporal data types (see Chapter 4) include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including subsecond divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an *anchored* time duration with a fixed starting point, such as the 10-day period from January 1, 2009, to January 10, 2009, inclusive).

Event Information versus Duration (or State) Information. A temporal data-base will store information concerning when certain events occur, or when certain facts are considered to be true. There are several different types of temporal information. **Point events** or **facts** are typically associated in the database with a **single time point** in some granularity. For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 2010). Note that even though such events or facts may have different granularities, each is still associated with a *single time value* in the database. This type of information is often represented as **time series data** as we will discuss in Section 26.2.5. **Duration events** or **facts**, on the other hand, are associated with a specific **time period** in the data-base. For example, an employee may have worked in a company from August 15, 2003 until November 20, 2008.

A **time period** is represented by its **start** and **end time points** [START-TIME, END-TIME]. For example, the above period is represented as [2003-08-15, 2008-11-20]. Such a time period is often interpreted to mean the *set of all time points* from start-time to end-time, inclusive, in the specified granularity. Hence, assuming day granularity, the period [2003-08-15, 2008-11-20] represents the set of all days from August 15, 2003, until November 20, 2008, inclusive.

Valid Time and Transaction Time Dimensions. Given a particular event or fact that is associated with a particular time point or time period in the database, the association may be interpreted to mean different things. The most natural interpretation is that the associated time is the time that the event occurred, or the period during which the fact was considered to be true *in the real world.* If this interpretation is used, the associated time is often referred to as the **valid time**. A temporal database using this interpretation is called a **valid time database**.

However, a different interpretation can be used, where the associated time refers to the time when the information was actually stored in the database; that is, it is the value of the system time clock when the information is valid *in the system.* In this case, the associated time is called the **transaction time**. A temporal database using this interpretation is called a **transaction time database**.

Other interpretations can also be intended, but these are considered to be the most common ones, and they are referred to as **time dimensions**. In some applications, only one of the dimensions is needed and in other cases both time dimensions are required, in which case the temporal database is called a **bitemporal database**. If other interpretations are intended for time, the user can define the semantics and program the applications appropriately, and it is called a **user-defined time**.

The next section shows how these concepts can be incorporated into relational databases, and Section 3 shows an approach to incorporate temporal concepts into object databases.

2. Incorporating Time in Relational Databases Using Tuple Versioning

Valid Time Relations. Let us now see how the different types of temporal data-bases may be represented in the relational model. First, suppose that we would like to include the history of changes as they occur in the real world. Consider again the database in Figure 26.1, and let us assume that, for this application, the granularity is day. Then, we could convert the two relations EMPLOYEE and DEPARTMENT into **valid time relations** by adding the attributes Vst (Valid Start Time) and Vet (Valid End Time), whose data type is DATE in order to provide day granularity. This is shown in Figure 26.7(a), where the relations have been renamed EMP_VT and DEPT_VT, respectively.

Consider how the EMP_VT relation differs from the nontemporal EMPLOYEE relation (Figure). InEMP_VT, each tuple V represents a **version** of an employee's

Information that is valid (in the real world) only during the time period [V.Vst, V.Vet], whereas in EMPLOYEE each tuple represents only the current state or current version of each employee. In EMP_VT, the **current version** of each employee typically has a special value, *now*, as its valid end time. This special value, ***now***, is a **temporal variable** that implicitly represents the current time as time progresses. The nontemporal EMPLOYEE relation would only include those tuples from the EMP_VT relation whose Vet is *now*.

Figure 26.8 shows a few tuple versions in the valid-time relations EMP_VT and DEPT_VT. There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan. We can now see how a valid time relation should behave when information is changed. Whenever one or more attributes of an employee are **updated**, rather than actually overwriting the old values, as would happen in a nontemporal relation, the system should create a new version and **close** the current version by changing its Vet to the end time. Hence, when the user issued the command to update the salary of Smith effective on June 1, 2003, to $30000, the second version of Smith was created (see Figure below). At the time of this update, the first version of Smith was the current version, with *now* as its Vet, but after the update *now* was changed to May 31, 2003 (one less than June 1, 2003, in day granularity), to indicate that the version has become a **closed** or **history version** and that the new (second) version of Smith is now the current one.

EMP_VT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31
Smith	123456789	30000	5	333445555	2003-06-01	Now
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31
Wong	333445555	40000	5	888665555	2002-04-01	Now
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10
Narayan	666884444	38000	5	333445555	2003-08-01	Now

. . .

DEPT_VT

Dname	Dno	Manager_ssn	Vst	Vet
Research	5	888665555	2001-09-20	2002-03-31
Research	5	333445555	2002-04-01	Now

It is important to note that in a valid time relation, the user must generally provide the valid time of an update. For example, the salary update of Smith may have been entered in the database on May 15, 2003, at 8:52:12 A.M., say, even though the salary change in the real world is effective on June 1, 2003. This is called a **proactive update**, since it is applied to the database *before* it becomes effective in the real world. If the update is applied to the database *after* it becomes effective in the real world, it is called a **retroactive update**. An update that is applied at the same time as it becomes effective is called a **simultaneous update**.

The action that corresponds to **deleting** an employee in a nontemporal database would typically be applied to a valid time database by *closing the current version* of the employee being deleted. For example, if Smith leaves the company effective January 19, 2004, then this would be applied by changing Vet of the current version of Smith from *now* to 2004-01-19. In this Figure, there is no current version for Brown, because he presumably left the company on 2002-08-10 and was *logically deleted.* However, because the database is temporal, the old information on Brown is still there.

The operation to **insert** a new employee would correspond to *creating the first tuple version* for that employee, and making it the current version, with the Vst being the effective (real world) time when the employee starts work. In previous Figure, the tuple on Narayan illustrates this, since the first version has not been updated yet.

Notice that in a valid time relation, the *nontemporal key*, such as Ssn in EMPLOYEE, is no longer unique in each tuple (version). The new relation key for EMP_VT is a combination of the nontemporal key and the valid start time attribute Vst, so we use (Ssn, Vst) as primary key. This is because, at any point in time, there should be *at most one valid version* of each entity. Hence, the constraint that any two tuple ver-sions representing the same entity should have *nonintersecting valid time periods* should hold on valid time relations. Notice that if the nontemporal primary key value may change over time, it is important to have a unique **surrogate key attribute**, whose value never changes for each real-world entity, in order to relate all versions of the same real-world entity.

Valid time relations basically keep track of the history of changes as they become effective in the *real world.* Hence, if all real-world changes are applied, the database keeps a history of the *real-world states* that are represented. However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual *database state* at any point in time. If the actual database states are important to an application, then one should use *transaction time relations.*

Transaction Time Relations. In a transaction time database, whenever a change is applied to the database, the actual **timestamp** of the transaction that applied the change (insert, delete, or update) is recorded. Such a database is most useful when changes are applied *simultaneously* in the majority of cases—for example, real-time stock trading or banking transactions. If we convert the nontemporal database in Figure 26.1 into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into **transaction time relations** by adding the attributes Tst (Transaction Start Time) and Tet (Transaction End Time), whose data type is typically TIMESTAMP. This is shown in previous Figure, where the relations have been renamed EMP_TT and DEPT_TT, respectively.

In EMP_TT, each tuple *V* represents a *version* of an employee's information that was created at actual time*V*.Tst and was (logically) removed at actual time *V*.Tet (because the information was no longer correct). InEMP_TT, the *current version* of each employee typically has a special value, ***uc*** **(Until Changed)**, as its transaction end time, which indicates that the tuple represents correct information *until it is changed* by some other transaction. A transaction time database has also been called a **rollback database**, because a user can logically roll back to the actual database state at any past point in time *T* by retrieving all tuple versions *V* whose transaction time period [*V*.Tst, *V*.Tet] includes time point *T*.

Bitemporal Relations. Some applications require both valid time and transaction time, leading to **bitemporal relations**. In our example, Figure 26.7(c) shows how the EMPLOYEE and DEPARTMENT nontemporal relations in Figure would appear as bitemporal relations EMP_BT and DEPT_BT, respectively. Above Figure shows a few tuples in these relations. In these tables, tuples whose transaction end time Tet is *uc* are the ones representing currently valid information, whereas tuples whose Tet is an absolute timestamp are tuples that were valid until (just before) that timestamp. Hence, the tuples with *uc* in Previous Figure of this section correspond to the valid time tuples in Figure 26.7. The transaction start time attribute Tst in each tuple is the timestamp of the transaction that created that tuple.

Now consider how an **update operation** would be implemented on a bitemporal relation. In this model of bitemporal databases, *no attributes are physically changed* in any tuple except for the transaction end time attribute Tet with a value of *uc*. To illustrate how tuples are created, consider the EMP_BT relation. The *current version V* of an employee has *uc* in its Tet attribute and *now* in its Vet attribute. If some attribute—say, Salary—is updated, then the transaction *T* that performs the update should have two parameters: the new value of Salary and the valid time VT when the new salary becomes effective (in the real world).

Assume that VT− is the

EMP_BT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet	Tst	Tet
Smith	123456789	25000	5	333445555	2002-06-15	Now	2002-06-08, 13:05:58	2003-06-04,08:56:12
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31	2003-06-04, 08:56:12	uc
Smith	123456789	30000	5	333445555	2003-06-01	Now	2003-06-04, 08:56:12	uc
Wong	333445555	25000	4	999887777	1999-08-20	Now	1999-08-20, 11:18:23	2001-01-07,14:33:02
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31	2001-01-07, 14:33:02	uc
Wong	333445555	30000	5	999887777	2001-02-01	Now	2001-01-07, 14:33:02	2002-03-28,09:23:57
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31	2002-03-28, 09:23:57	uc
Wong	333445555	40000	5	888667777	2002-04-01	Now	2002-03-28, 09:23:57	uc
Brown	222447777	28000	4	999887777	2001-05-01	Now	2001-04-27, 16:22:05	2002-08-12,10:11:07
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10	2002-08-12, 10:11:07	uc
Narayan	666884444	38000	5	333445555	2003-08-01	Now	2003-07-28, 09:25:37	uc

...

DEPT_VT

Dname	Dno	Manager_ssn	Vst	Vet	Tst	Tet
Research	5	888665555	2001-09-20	Now	2001-09-15,14:52:12	2001-03-28,09:23:57
Research	5	888665555	2001-09-20	1997-03-31	2002-03-28,09:23:57	uc
Research	5	333445555	2002-04-01	Now	2002-03-28,09:23:57	uc

time point before VT in the given valid time granularity and that transaction *T* has a timestamp TS(*T*). Then, the following physical changes would be applied to the

EMP_BT table:

1. Make a copy V_2 of the current version *V*; set V_2.Vet to VT−, V_2.Tst to TS(*T*), V_2.Tet to *uc,* and insert V_2in EMP_BT; V_2 is a copy of the previous current version *V after it is closed* at valid time VT−.

2. Make a copy V_3 of the current version *V*; set V_3.Vst to VT, V_3.Vet to *now,* V_3.Salary to the new salary value, V_3.Tst to TS(*T*), V_3.Tet to *uc,* and insert V_3 in EMP_BT; V_3 represents the new current version.

3. Set *V*.Tet to TS(*T*) since the current version is no longer representing correct information.

As an illustration, consider the first three tuples V_1, V_2, and V_3 in EMP_BT in Figure 26.9. Before the update of Smith's salary from 25000 to 30000, only V_1 was in EMP_BT and it was the current version and its Tetwas *uc.* Then, a transaction *T* whose timestamp TS(*T*) is '2003-06-04,08:56:12' updates the salary to 30000with the effective valid time of '2003-06-01'. The tuple V_2 is created, which is a copy of V_1 except that itsVet is set to '2003-05-31', one day less than the new valid time and its Tst is the timestamp of the updating transaction. The tuple V_3 is also created, which has the new salary, its Vst is set to '2003-06-01', and its Tstis also the timestamp of the updating transaction. Finally, the Tet of V_1 is set to the timestamp of the updating transaction, '2003-06-04,08:56:12'. Note that this is a *retroactive update,* since the updating transaction ran on June 4, 2003, but the salary change is effective on June 1, 2003.

Similarly, when Wong's salary and department are updated (at the same time) to 30000 and 5, the updating transaction's timestamp is '2001-01-07,14:33:02' and the effective valid time for the update is '2001-02-01'. Hence, this is a *proactive update* because the transaction ran on January 7, 2001, but the effective date was February 1, 2001. In this case, tuple V_4 is logically replaced by V_5 and V_6.

Next, let us illustrate how a **delete operation** would be implemented on a bitemporal relation by considering the tuples V_9 and V_{10} in the EMP_BT relation of Figure 26.9. Here, employee Brown left the company effective August 10, 2002, and the logical delete is carried out by a transaction T with TS(T) = 2002-08-12,10:11:07. Before this, V9 was the current version of Brown, and its Tet was *uc*. The logical delete is implemented by setting V_9.Tet to 2002-08-12,10:11:07 to invalidate it, and creating the *final version* V_{10} for Brown, with its Vet = 2002-08-10 (see Figure 26.9). Finally, an **insert operation** is implemented by creating the *first version* as illustrated by V_{11} in the EMP_BT table.

Implementation Considerations. There are various options for storing the tuples in a temporal relation. One is to store all the tuples in the same table, as shown in Figures 26.8 and 26.9. Another option is to create two tables: one for the currently valid information and the other for the rest of the tuples. For example, in the bitemporal EMP_BT relation, tuples with *uc* for their Tet and *now* for their Vet would be in one relation, the *current table,* since they are the ones currently valid (that is, represent the current snapshot), and all other tuples would be in another relation. This allows the database administrator to have different access paths, such as indexes for each relation, and keeps the size of the current table reasonable. Another possibility is to create a third table for corrected tuples whose Tet is not *uc.*

Another option that is available is to *vertically partition* the attributes of the temporal relation into separate relations so that if a relation has many attributes, a whole new tuple version is created whenever any one of the attributes is updated. If the attributes are updated asynchronously, each new version may differ in only one of the attributes, thus needlessly repeating the other attribute values. If a separate relation is created to contain only the attributes that *always change synchronously,* with the primary key replicated in each relation, the database is said to be in **temporal normal form**. However, to combine the information, a variation of join known as **temporal intersection join** would be needed, which is generally expensive to implement.

It is important to note that bitemporal databases allow a complete record of changes. Even a record of corrections is possible. For example, it is possible that two tuple versions of the same employee may have the same valid time but different attribute values as long as their transaction times are disjoint. In this case, the tuple with the later transaction time is a **correction** of the other tuple version. Even incorrectly entered valid times may be corrected this way. The incorrect state of the data base will still be available as a previous database state for querying purposes. A data-base that keeps such a complete record of changes and corrections is sometimes called an **append-only database**.

3. Incorporating Time in Object-Oriented Databases Using Attribute Versioning

The previous section discussed the **tuple versioning approach** to implementing temporal databases. In this approach, whenever one attribute value is changed, a whole new tuple version is created, even though all the other attribute values will be identical to the previous tuple version. An alternative approach can be used in database systems that support **complex structured objects**, such as object data-bases or object-relational systems. This approach is called **attribute versioning**.

In attribute versioning, a single complex object is used to store all the temporal changes of the object. Each attribute that changes over time is called a **time-varying attribute**, and it has its values versioned over time by adding temporal periods to the attribute. The temporal periods may represent valid time, transaction time, or bitemporal, depending on the application requirements. Attributes that do not change over time are called **nontime-varying** and are not associated with the temporal periods. To illustrate this, consider the example in Figure 26.10, which is an attribute-versioned valid time representation of EMPLOYEE using the object definition language (ODL) notation for object databases (see Chapter 11). Here, we assumed that name and Social Security number are nontime-varying attributes, whereas salary, department, and supervisor are time-varying attributes (they may change over time). Each time-varying attribute is represented as a list of tuples

<Valid_start_time, Valid_end_time, Value>, ordered by valid start time.

Whenever an attribute is changed in this model, the current attribute version is *closed* and a **new attribute version** for this attribute only is appended to the list. This allows attributes to change asynchronously. The current value for each attribute has *now* for its Valid_end_time. When using attribute versioning, it is useful to include a **lifespan temporal attribute** associated with the whole object whose value is one or more valid time periods that indicate the valid time of existence for the whole object. Logical deletion of the object is implemented by closing the lifespan. The constraint that any time period of an attribute within an object should be a subset of the object's lifespan should be enforced.

For bitemporal databases, each attribute version would have a tuple with five components:

<Valid_start_time, Valid_end_time, Trans_start_time, Trans_end_time, Value>

The object lifespan would also include both valid and transaction time dimensions. Therefore, the full capabilities of bitemporal databases can be available with attribute versioning. Mechanisms similar to those discussed earlier for updating tuple versions can be applied to updating attribute versions.

```
class TEMPORAL_SALARY
{       attribute     Date  Valid_start_time;
        attribute     Date  Valid_end_time;
        attribute     float  Salary;
};
class TEMPORAL_DEPT
{       attribute     Date  Valid_start_time;
        attribute     Date  Valid_end_time;
        attribute     DEPARTMENT_VT     Dept;
};
class TEMPORAL_SUPERVISOR
{       attribute     Date  Valid_start_time;
        attribute     Date  Valid_end_time;
        attribute     EMPLOYEE_VT Supervisor;
};
class TEMPORAL_LIFESPAN
{       attribute     Date  Valid_ start time;
        attribute     Date  Valid end time;
};
class EMPLOYEE_VT
(       extent EMPLOYEES  )
{       attribute     list<TEMPORAL_LIFESPAN>       lifespan;
        attribute     string          Name;
        attribute     string          Ssn;
        attribute     list<TEMPORAL_SALARY> Sal_history;
        attribute     list<TEMPORAL_DEPT>      Dept_history;
        attribute     list <TEMPORAL_SUPERVISOR> Supervisor_history;
};
```

4. Temporal Querying Constructs and the TSQL2 Language

So far, we have discussed how data models may be extended with temporal con-structs. Now we give a brief overview of how query operations need to be extended for temporal querying. We will briefly discuss the TSQL2 language, which extends SQL for querying valid time, transaction time, and bitemporal relational databases.

In nontemporal relational databases, the typical selection conditions involve attribute conditions, and tuples that satisfy these conditions are selected from the set of *current tuples.* Following that, the attributes of interest to the query are specified by a *projection operation* (see Chapter 6). For example, in the query to retrieve the names of all employees working in department 5 whose salary is greater than 30000, the selection condition would be as follows:

((Salary > 30000) AND (Dno = 5))

The projected attribute would be Name. In a temporal database, the conditions may involve time in addition to attributes. A **pure time condition** involves only time— for example, to select all employee tuple versions that were valid on a certain *time point* T or that were valid *during a certain time period* [T_1, T_2]. In this case, the specified time period is compared with the valid time period of each tuple version [T.Vst, T.Vet], and only those tuples that satisfy the condition are selected. In these operations, a period is considered to be equivalent to the set of time points from T_1 to T_2 inclusive, so the standard set comparison operations can be used. Additional operations, such as whether one time period ends *before* another starts are also needed.

Some of the more common operations used in queries are as follows:

[T.Vst, T.Vet] **INCLUDES** [T_1, T_2]	Equivalent to $T_1 \geq T$.Vst AND $T_2 \leq T$.Vet
[T.Vst, T.Vet] **INCLUDED_IN** [T_1, T_2]	Equivalent to $T_1 \leq T$.Vst AND $T_2 \geq T$.Vet
[T.Vst, T.Vet] **OVERLAPS** [T_1, T_2]	Equivalent to ($T_1 \leq T$.Vet AND $T_2 \geq T$.Vst)[22]
[T.Vst, T.Vet] **BEFORE** [T_1, T_2]	Equivalent to $T_1 \geq T$.Vet
[T.Vst, T.Vet] **AFTER** [T_1, T_2]	Equivalent to $T_2 \leq T$.Vst
[T.Vst, T.Vet] **MEETS_BEFORE** [T_1, T_2]	Equivalent to $T_1 = T$.Vet + 1[23]
[T.Vst, T.Vet] **MEETS_AFTER** [T_1, T_2]	Equivalent to $T_2 + 1 = T$.Vst

Additionally, operations are needed to manipulate time periods, such as computing the union or intersection of two time periods. The results of these operations may not themselves be periods, but rather **temporal elements**—a collection of one or more *disjoint* time periods such that no two time periods in a temporal element are directly adjacent. That is, for any two time periods [T_1, T_2] and [T_3, T_4] in a temporal element, the following three conditions must hold:

[T_1, T_2] intersection [T_3, T_4] is empty.

T_3 is not the time point following T_2 in the given granularity.

T_1 is not the time point following T_4 in the given granularity.

The latter conditions are necessary to ensure unique representations of temporal elements. If two time periods [T_1, T_2] and [T_3, T_4] are adjacent, they are combined into a single time period [T_1, T_4]. This is called **coalescing** of time periods. Coalescing also combines intersecting time periods.

To illustrate how pure time conditions can be used, suppose a user wants to select all employee versions that were valid at any point during 2002. The appropriate selection condition applied to the relation in Figure would be

[*T*.Vst, *T*.Vet] OVERLAPS [2002-01-01, 2002-12-31]

Typically, most temporal selections are applied to the valid time dimension. For a bitemporal database, one usually applies the conditions to the currently correct tuples with *uc* as their transaction end times. However, if the query needs to be applied to a previous database state, an AS_OF *T* clause is appended to the query, which means that the query is applied to the valid time tuples that were correct in the database at time *T*.

In addition to pure time conditions, other selections involve **attribute and time conditions**. For example, suppose we wish to retrieve all EMP_VT tuple versions T for employees who worked in department 5 at any time during 2002. In this case, the condition is

[*T*.Vst, *T*.Vet]OVERLAPS [2002-01-01, 2002-12-31] AND (*T*.Dno = 5)

Finally, we give a brief overview of the TSQL2 query language, which extends SQL with constructs for temporal databases. The main idea behind TSQL2 is to allow users to specify whether a relation is nontemporal (that is, a standard SQL relation) or temporal. The CREATE TABLE statement is extended with an *optional* AS clause to allow users to declare different temporal options. The following options are avail-able:

<AS VALID STATE <GRANULARITY> (valid time relation with valid time period)

<AS VALID EVENT <GRANULARITY> (valid time relation with valid time point)

<AS TRANSACTION (transaction time relation with transaction time period)

<AS VALID STATE <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time period)

<AS VALID EVENT <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time point)

The Keywords STATE and EVENT are used to specify whether a time *period* or time *point* is associated with the valid time dimension. In TSQL2, rather than have the user actually see how the temporal tables are implemented, the TSQL2 language adds query language constructs to specify various types of temporal selections, temporal projections, temporal aggregations, transformation among granularities, and many other concepts.

5. Time Series Data

Time series data is used very often in financial, sales, and economics applications. They involve data values that are recorded according to a specific predefined sequence of time points. Therefore, they are a special type of **valid event data**, where the event time points are predetermined according to a fixed calendar. Consider the example of closing daily stock prices of a particular company on the New York Stock Exchange. The granularity here is day, but the days that the stock market is open are known (non holiday weekdays). Hence, it has been common to specify a computational procedure that calculates the particular **calendar** associated with a time series. Typical queries on time series involve **temporal aggregation** over higher granularity intervals—for example, finding the average or maximum *weekly* closing stock price or the maximum and minimum *monthly* closing stock price from the *daily* information.

As another example, consider the daily sales dollar amount at each store of a chain of stores owned by a particular company. Again, typical temporal aggregates would be retrieving the weekly, monthly, or yearly sales from the daily sales information (using the sum aggregate function), or comparing same store monthly sales with previous monthly sales, and so on.

Because of the specialized nature of time series data and the lack of support for it in older DBMSs, it has been common to use specialized **time series management systems** rather than general-purpose DBMSs for managing such information. In such systems, it has been common to store time series values in sequential order in a file, and apply specialized time series procedures to analyze the information. The problem with this approach is that the full power of high-level querying in languages such as SQL will not be available in such systems.

More recently, some commercial DBMS packages are offering time series extensions, such as the Oracle time cartridge and the time series data blade of Informix Universal Server. In addition, the TSQL2 language provides some support for time series in the form of event tables.

Distributed Database Management: Reference Architecture, levels of distribution transparency, Distributed database design, Distributed Query Processing and Optimization, Distributed Transaction Modeling.

Distributed Database:

In a distributed database, there are a number of databases that may be geographically distributed all over the world. A distributed DBMS manages the distributed database in a manner so that it appears as one single database to users. In the later part of the chapter, we go on to study the factors that lead to distributed databases, its advantages and disadvantages.

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

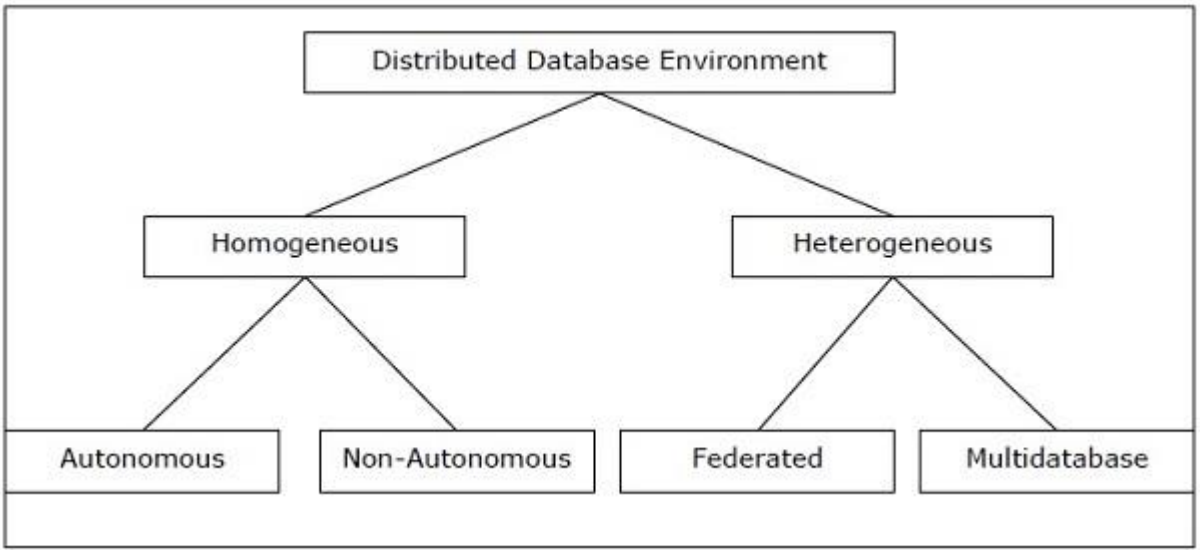
Following are some of the adversities associated with distributed databases.

- **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.

This part starts with the types of distributed databases. Distributed databases can be classified into homogeneous and heterogeneous databases having further divisions. The next section of this chapter discusses the distributed architectures namely client – server, peer – to – peer and multi – DBMS. Finally, the different design alternatives like replication and fragmentation are introduced.

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 −

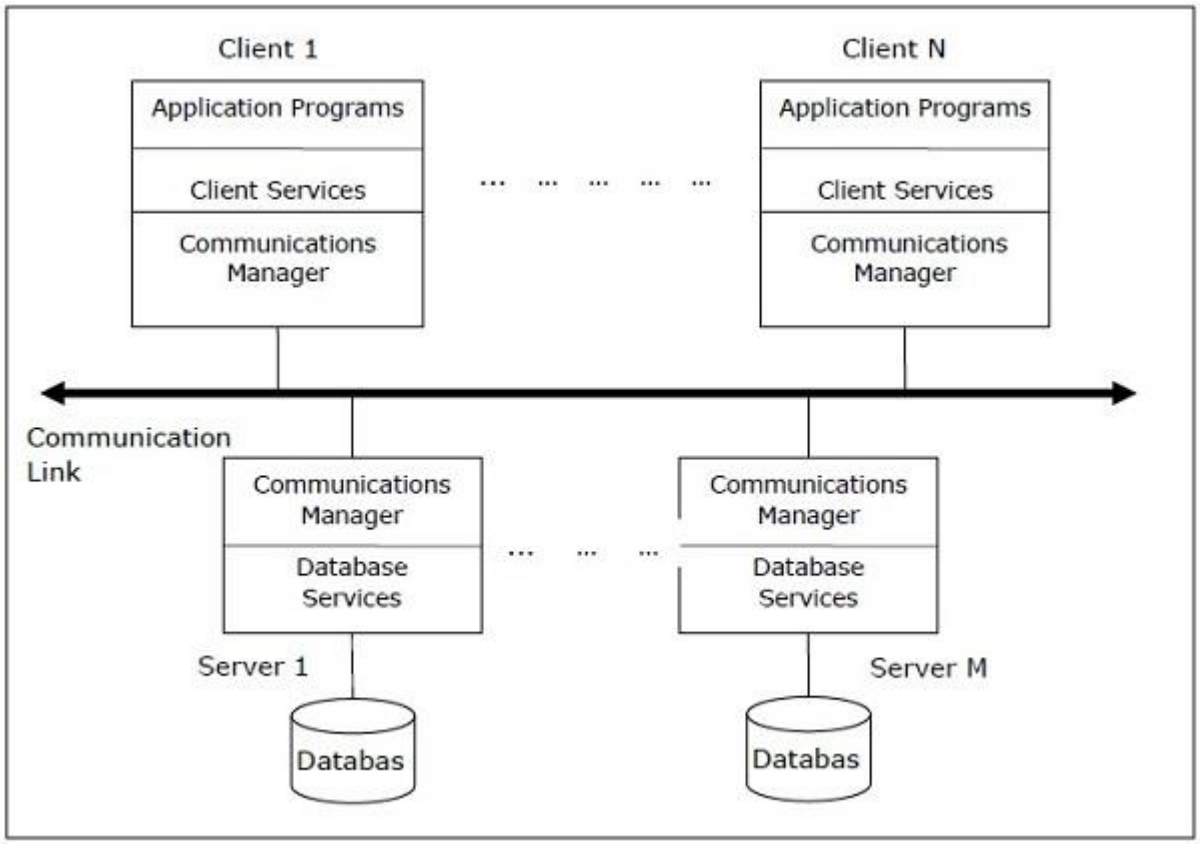
- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

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)

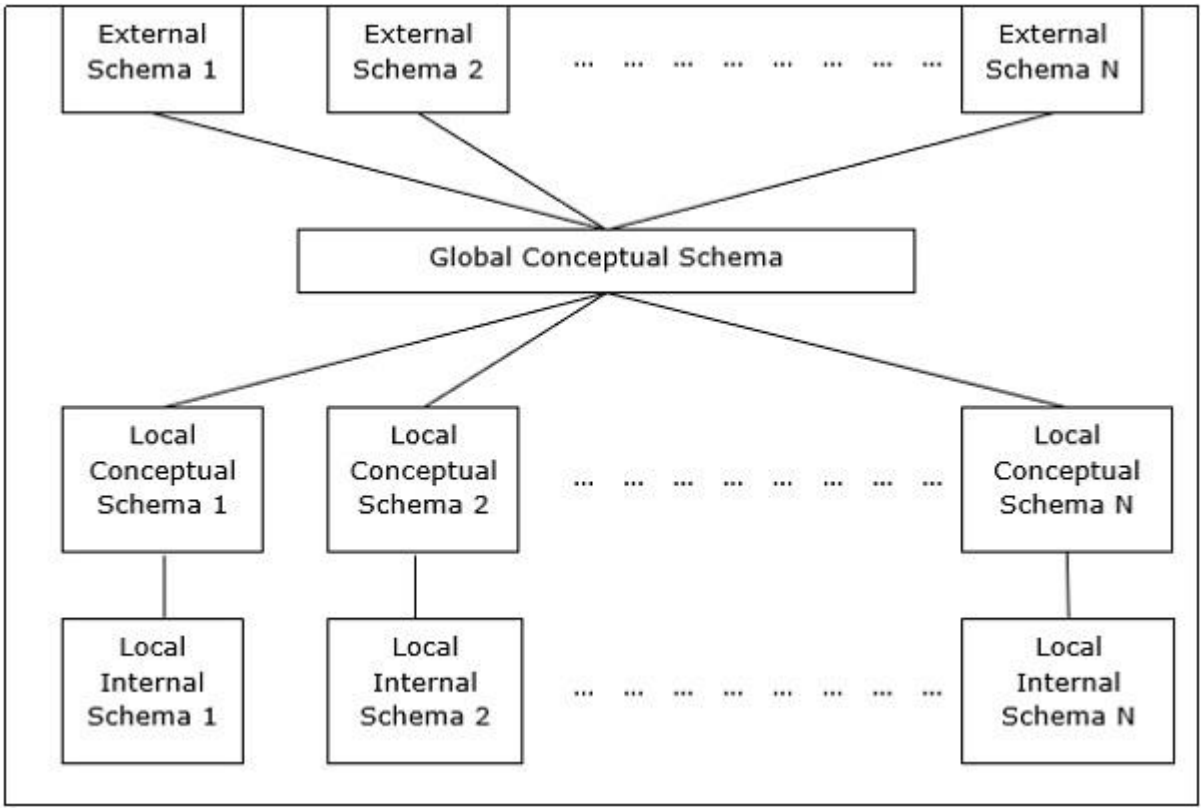


Peer- to-Peer Architecture for DDBMS

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

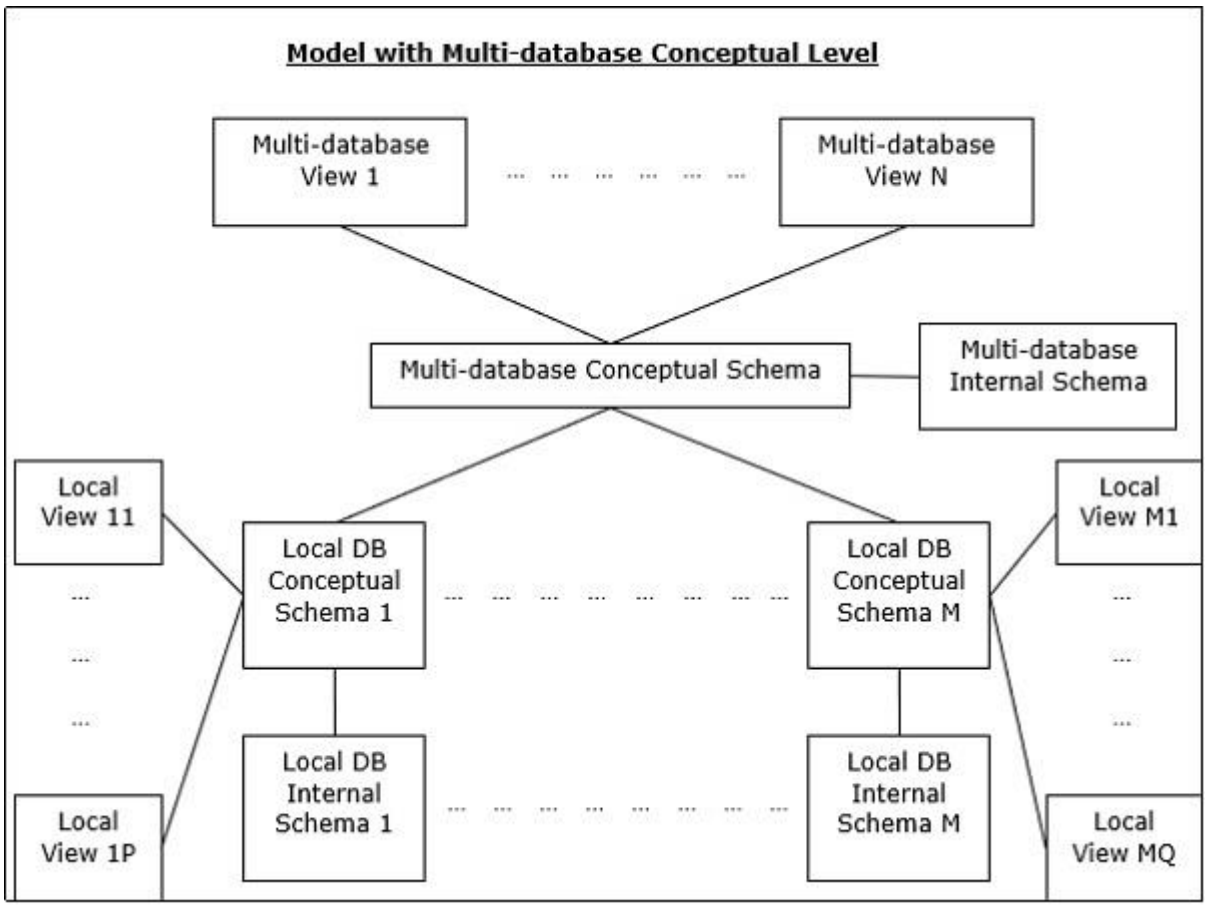
This is an integrated database system formed by a collection of two or more autonomous database systems.

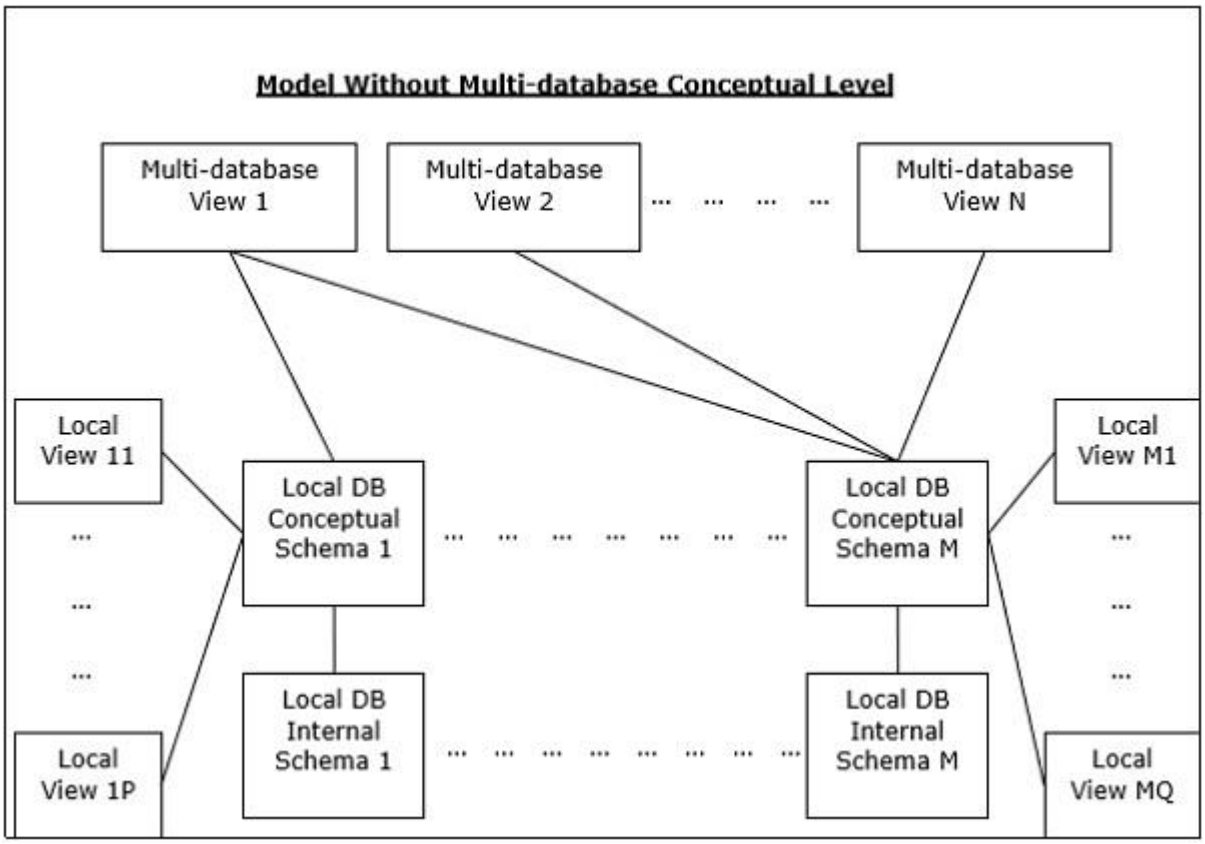
Multi-DBMS can be expressed through six levels of schemas −

- **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.





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.

Data Replication

Data replication is the process of storing separate copies of the database at two or more sites. It is a popular fault tolerance technique of distributed databases.

Advantages of Data Replication

- **Reliability** − In case of failure of any site, the database system continues to work since a copy is available at another site(s).
- **Reduction in Network Load** − Since local copies of data are available, query processing can be done with reduced network usage, particularly during prime hours. Data updating can be done at non-prime hours.
- **Quicker Response** − Availability of local copies of data ensures quick query processing and consequently quick response time.
- **Simpler Transactions** − Transactions require less number of joins of tables located at different sites and minimal coordination across the network. Thus, they become simpler in nature.

Disadvantages of Data Replication

- **Increased Storage Requirements** − Maintaining multiple copies of data is associated with increased storage costs. The storage space required is in multiples of the storage required for a centralized system.

- **Increased Cost and Complexity of Data Updating** − Each time a data item is updated, the update needs to be reflected in all the copies of the data at the different sites. This requires complex synchronization techniques and protocols.
- **Undesirable Application – Database coupling** − If complex update mechanisms are not used, removing data inconsistency requires complex co-ordination at application level. This results in undesirable application – database coupling.

Some commonly used replication techniques are −

- Snapshot replication
- Near-real-time replication
- Pull replication

Fragmentation

Fragmentation is the task of dividing a table into a set of smaller tables. The subsets of the table are called **fragments**. Fragmentation can be of three types: horizontal, vertical, and hybrid (combination of horizontal and vertical). Horizontal fragmentation can further be classified into two techniques: primary horizontal fragmentation and derived horizontal fragmentation.

Fragmentation should be done in a way so that the original table can be reconstructed from the fragments. This is needed so that the original table can be reconstructed from the fragments whenever required. This requirement is called "reconstructiveness."

Advantages of Fragmentation

- Since data is stored close to the site of usage, efficiency of the database system is increased.
- Local query optimization techniques are sufficient for most queries since data is locally available.
- Since irrelevant data is not available at the sites, security and privacy of the database system can be maintained.

Disadvantages of Fragmentation

- When data from different fragments are required, the access speeds may be very high.
- In case of recursive fragmentations, the job of reconstruction will need expensive techniques.
- Lack of back-up copies of data in different sites may render the database ineffective in case of failure of a site.

Vertical Fragmentation

In vertical fragmentation, the fields or columns of a table are grouped into fragments. In order to maintain reconstructiveness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

For example, let us consider that a University database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

Now, the fees details are maintained in the accounts section. In this case, the designer will fragment the database as follows −

```
CREATE TABLE STD_FEES AS
   SELECT Regd_No, Fees
   FROM STUDENT;
```

Horizontal Fragmentation

Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructiveness. Each horizontal fragment must have all columns of the original base table.

For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows −

```
CREATE COMP_STD AS
   SELECT * FROM STUDENT
   WHERE COURSE = "Computer Science";
```

Hybrid Fragmentation

In hybrid fragmentation, a combination of horizontal and vertical fragmentation techniques are used. This is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

Hybrid fragmentation can be done in two alternative ways −

- At first, generate a set of horizontal fragments; then generate vertical fragments from one or more of the horizontal fragments.
- At first, generate a set of vertical fragments; then generate horizontal fragments from one or more of the vertical fragments.

Distribution transparency is the property of distributed databases by the virtue of which the internal details of the distribution are hidden from the users. The DDBMS designer may choose to fragment tables, replicate the fragments and store them at different sites. However, since users are oblivious of these details, they find the distributed database easy to use like any centralized database.

The three dimensions of distribution transparency are −

- Location transparency

- Fragmentation transparency
- Replication transparency

Location Transparency

Location transparency ensures that the user can query on any table(s) or fragment(s) of a table as if they were stored locally in the user's site. The fact that the table or its fragments are stored at remote site in the distributed database system, should be completely oblivious to the end user. The address of the remote site(s) and the access mechanisms are completely hidden.

In order to incorporate location transparency, DDBMS should have access to updated and accurate data dictionary and DDBMS directory which contains the details of locations of data.

Fragmentation Transparency

Fragmentation transparency enables users to query upon any table as if it were unfragmented. Thus, it hides the fact that the table the user is querying on is actually a fragment or union of some fragments. It also conceals the fact that the fragments are located at diverse sites.

This is somewhat similar to users of SQL views, where the user may not know that they are using a view of a table instead of the table itself.

Replication Transparency

Replication transparency ensures that replication of databases are hidden from the users. It enables users to query upon a table as if only a single copy of the table exists.

Replication transparency is associated with concurrency transparency and failure transparency. Whenever a user updates a data item, the update is reflected in all the copies of the table. However, this operation should not be known to the user. This is concurrency transparency. Also, in case of failure of a site, the user can still proceed with his queries using replicated copies without any knowledge of failure. This is failure transparency.

Combination of Transparencies

In any distributed database system, the designer should ensure that all the stated transparencies are maintained to a considerable extent. The designer may choose to fragment tables, replicate them and store them at different sites; all oblivious to the end user. However, complete distribution transparency is a tough task and requires considerable design efforts.

Database control refers to the task of enforcing regulations so as to provide correct data to authentic users and applications of a database. In order that correct data is available to users, all data should conform to the integrity constraints defined in the database. Besides, data should be screened away from unauthorized users so as to maintain security and privacy of the database. Database control is one of the primary tasks of the database administrator (DBA).

The three dimensions of database control are −

- Authentication
- Access rights
- Integrity constraints

Authentication

In a distributed database system, authentication is the process through which only legitimate users can gain access to the data resources.

Authentication can be enforced in two levels −

- **Controlling Access to Client Computer** − At this level, user access is restricted while login to the client computer that provides user-interface to the database server. The most common method is a username/password combination. However, more sophisticated methods like biometric authentication may be used for high security data.
- **Controlling Access to the Database Software** − At this level, the database software/administrator assigns some credentials to the user. The user gains access to the database using these credentials. One of the methods is to create a login account within the database server.

Access Rights

A user's access rights refers to the privileges that the user is given regarding DBMS operations such as the rights to create a table, drop a table, add/delete/update tuples in a table or query upon the table.

In distributed environments, since there are large number of tables and yet larger number of users, it is not feasible to assign individual access rights to users. So, DDBMS defines certain roles. A role is a construct with certain privileges within a database system. Once the different roles are defined, the individual users are assigned one of these roles. Often a hierarchy of roles are defined according to the organization's hierarchy of authority and responsibility.

For example, the following SQL statements create a role "Accountant" and then assigns this role to user "ABC".

```
CREATE ROLE ACCOUNTANT;
GRANT SELECT, INSERT, UPDATE ON EMP_SAL TO ACCOUNTANT;
GRANT INSERT, UPDATE, DELETE ON TENDER TO ACCOUNTANT;
GRANT INSERT, SELECT ON EXPENSE TO ACCOUNTANT;
COMMIT;
GRANT ACCOUNTANT TO ABC;
COMMIT;
```

Semantic Integrity Control

Semantic integrity control defines and enforces the integrity constraints of the database system.

The integrity constraints are as follows −

- Data type integrity constraint
- Entity integrity constraint
- Referential integrity constraint

Data Type Integrity Constraint

A data type constraint restricts the range of values and the type of operations that can be applied to the field with the specified data type.

For example, let us consider that a table "HOSTEL" has three fields - the hostel number, hostel name and capacity. The hostel number should start with capital letter "H" and cannot be NULL, and the capacity should not be more than 150. The following SQL command can be used for data definition −

```
CREATE TABLE HOSTEL (
   H_NO VARCHAR2(5) NOT NULL,
   H_NAME VARCHAR2(15),
   CAPACITY INTEGER,
   CHECK ( H_NO LIKE 'H%'),
   CHECK ( CAPACITY <= 150)
);
```

Entity Integrity Control

Entity integrity control enforces the rules so that each tuple can be uniquely identified from other tuples. For this a primary key is defined. A primary key is a set of minimal fields that can uniquely identify a tuple. Entity integrity constraint states that no two tuples in a table can have identical values for primary keys and that no field which is a part of the primary key can have NULL value.

For example, in the above hostel table, the hostel number can be assigned as the primary key through the following SQL statement (ignoring the checks) −

```
CREATE TABLE HOSTEL (
   H_NO VARCHAR2(5) PRIMARY KEY,
   H_NAME VARCHAR2(15),
   CAPACITY INTEGER
);
```

Referential Integrity Constraint

Referential integrity constraint lays down the rules of foreign keys. A foreign key is a field in a data table that is the primary key of a related table. The referential integrity constraint lays down the rule that the value of the foreign key field should either be among the values of the primary key of the referenced table or be entirely NULL.

For example, let us consider a student table where a student may opt to live in a hostel. To include this, the primary key of hostel table should be included as a foreign key in the student table. The following SQL statement incorporates this –

```
CREATE TABLE STUDENT (
   S_ROLL INTEGER PRIMARY KEY,
   S_NAME VARCHAR2(25) NOT NULL,
   S_COURSE VARCHAR2(10),
   S_HOSTEL VARCHAR2(5) REFERENCES HOSTEL
);
```

When a query is placed, it is at first scanned, parsed and validated. An internal representation of the query is then created such as a query tree or a query graph. Then alternative execution strategies are devised for retrieving results from the database tables. The process of choosing the most appropriate execution strategy for query processing is called query optimization.

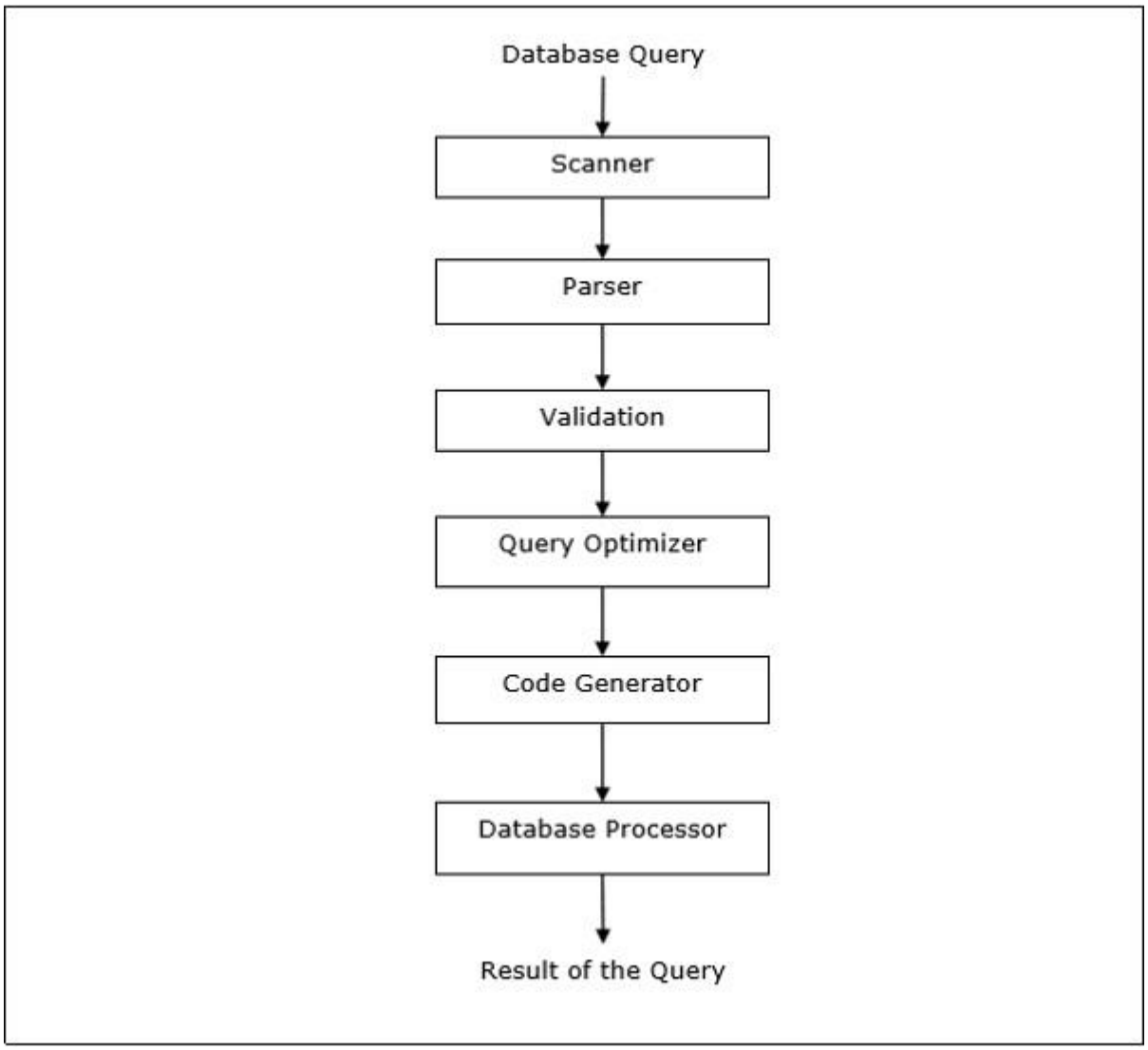
Query Optimization Issues in DDBMS

In DDBMS, query optimization is a crucial task. The complexity is high since number of alternative strategies may increase exponentially due to the following factors −

- The presence of a number of fragments.
- Distribution of the fragments or tables across various sites.
- The speed of communication links.
- Disparity in local processing capabilities.

Hence, in a distributed system, the target is often to find a good execution strategy for query processing rather than the best one. The time to execute a query is the sum of the following −

- Time to communicate queries to databases.
- Time to execute local query fragments.
- Time to assemble data from different sites.
- Time to display results to the application.



Relational Algebra

Relational algebra defines the basic set of operations of relational database model. A sequence of relational algebra operations forms a relational algebra expression. The result of this expression represents the result of a database query.

The basic operations are −

- Projection
- Selection
- Union
- Intersection
- Minus
- Join

Projection

Projection operation displays a subset of fields of a table. This gives a vertical partition of the table.

Syntax in Relational Algebra

$$\pi_{<{AttributeList}>}{(<{Table Name}>)}$$

For example, let us consider the following Student database −

STUDENT				
Roll_No	**Name**	**Course**	**Semester**	**Gender**
2	Amit Prasad	BCA	1	Male
4	Varsha Tiwari	BCA	1	Female
5	Asif Ali	MCA	2	Male
6	Joe Wallace	MCA	1	Male
8	Shivani Iyengar	BCA	1	Female

If we want to display the names and courses of all students, we will use the following relational algebra expression −

$$\pi_{Name,Course}{(STUDENT)}$$

Selection

Selection operation displays a subset of tuples of a table that satisfies certain conditions. This gives a horizontal partition of the table.

Syntax in Relational Algebra

$$\sigma_{<{Conditions}>}{(<{Table Name}>)}$$

For example, in the Student table, if we want to display the details of all students who have opted for MCA course, we will use the following relational algebra expression −

$$\sigma_{Course} = {\small "BCA"}^{(STUDENT)}$$

Combination of Projection and Selection Operations

For most queries, we need a combination of projection and selection operations. There are two ways to write these expressions −

- Using sequence of projection and selection operations.
- Using rename operation to generate intermediate results.

For example, to display names of all female students of the BCA course −

- Relational algebra expression using sequence of projection and selection operations

$$\pi_{Name}(\sigma_{Gender = \small "Female" AND \: Course = \small "BCA"}{(STUDENT)})$$

- Relational algebra expression using rename operation to generate intermediate results

$$FemaleBCAStudent \leftarrow \sigma_{Gender = \small "Female" AND \: Course = \small "BCA"} {(STUDENT)}$$

$$Result \leftarrow \pi_{Name}{(FemaleBCAStudent)}$$

Union

If P is a result of an operation and Q is a result of another operation, the union of P and Q ($p \cup Q$) is the set of all tuples that is either in P or in Q or in both without duplicates.

For example, to display all students who are either in Semester 1 or are in BCA course −

$$Sem1Student \leftarrow \sigma_{Semester = 1}{(STUDENT)}$$

$$BCAStudent \leftarrow \sigma_{Course = \small "BCA"}{(STUDENT)}$$

$$Result \leftarrow Sem1Student \cup BCAStudent$$

Intersection

If P is a result of an operation and Q is a result of another operation, the intersection of P and Q ($p \cap Q$) is the set of all tuples that are in P and Q both.

For example, given the following two schemas −

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

To display the names of all cities where a project is located and also an employee resides −

$$CityEmp \leftarrow \pi_{City}{(EMPLOYEE)}$$

$$CityProject \leftarrow \pi_{City}{(PROJECT)}$$

$$Result \leftarrow CityEmp \cap CityProject$$

Minus

If P is a result of an operation and Q is a result of another operation, P - Q is the set of all tuples that are in P and not in Q.

For example, to list all the departments which do not have an ongoing project (projects with status = ongoing) −

$$AllDept \leftarrow \pi_{Department}{(EMPLOYEE)}$$

$$ProjectDept \leftarrow \pi_{Department} (\sigma_{Status = \small "ongoing"}{(PROJECT)})$$

$$Result \leftarrow AllDept - ProjectDept$$

Join

Join operation combines related tuples of two different tables (results of queries) into a single table.

For example, consider two schemas, Customer and Branch in a Bank database as follows −

CUSTOMER

CustID	AccNo	TypeOfAc	BranchID	DateOfOpening

BRANCH

BranchID	BranchName	IFSCcode	Address

To list the employee details along with branch details −

$$Result \leftarrow CUSTOMER \bowtie_{Customer.BranchID=Branch.BranchID}{BRANCH}$$

Translating SQL Queries into Relational Algebra

SQL queries are translated into equivalent relational algebra expressions before optimization. A query is at first decomposed into smaller query blocks. These blocks are translated to equivalent relational algebra expressions. Optimization includes optimization of each block and then optimization of the query as a whole.

Examples

Let us consider the following schemas −

EMPLOYEE

EmpID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

WORKS

EmpID	PID	Hours

Example 1

To display the details of all employees who earn a salary LESS than the average salary, we write the SQL query −

```
SELECT * FROM EMPLOYEE
WHERE SALARY < ( SELECT AVERAGE(SALARY) FROM EMPLOYEE ) ;
```

This query contains one nested sub-query. So, this can be broken down into two blocks.

The inner block is −

```
SELECT AVERAGE(SALARY)FROM EMPLOYEE ;
```

If the result of this query is AvgSal, then outer block is −

```
SELECT * FROM EMPLOYEE WHERE SALARY < AvgSal;
```

Relational algebra expression for inner block −

$$AvgSal \leftarrow \Im_{AVERAGE(Salary)}{EMPLOYEE}$$

Relational algebra expression for outer block −

$$\sigma_{Salary <{AvgSal}>}{EMPLOYEE}$$

Example 2

To display the project ID and status of all projects of employee 'Arun Kumar', we write the SQL query −

```
SELECT PID, STATUS FROM PROJECT
WHERE PID = ( SELECT FROM WORKS  WHERE EMPID = ( SELECT EMPID
FROM EMPLOYEE
            WHERE NAME = 'ARUN KUMAR'));
```

This query contains two nested sub-queries. Thus, can be broken down into three blocks, as follows −

```
SELECT EMPID FROM EMPLOYEE WHERE NAME = 'ARUN KUMAR';
SELECT PID FROM WORKS WHERE EMPID = ArunEmpID;
SELECT PID, STATUS FROM PROJECT WHERE PID = ArunPID;
```

(Here ArunEmpID and ArunPID are the results of inner queries)

Relational algebra expressions for the three blocks are −

$$ArunEmpID \leftarrow \pi_{EmpID}(\sigma_{Name = \small "Arun Kumar"} {(EMPLOYEE)})$$

$$ArunPID \leftarrow \pi_{PID}(\sigma_{EmpID = \small "ArunEmpID"} {(WORKS)})$$

$$Result \leftarrow \pi_{PID, Status}(\sigma_{PID = \small "ArunPID"} {(PROJECT)})$$

Computation of Relational Algebra Operators

The computation of relational algebra operators can be done in many different ways, and each alternative is called an **access path**.

The computation alternative depends upon three main factors −

- Operator type
- Available memory
- Disk structures

The time to perform execution of a relational algebra operation is the sum of −

- Time to process the tuples.
- Time to fetch the tuples of the table from disk to memory.

Since the time to process a tuple is very much smaller than the time to fetch the tuple from the storage, particularly in a distributed system, disk access is very often considered as the metric for calculating cost of relational expression.

Computation of Selection

Computation of selection operation depends upon the complexity of the selection condition and the availability of indexes on the attributes of the table.

Following are the computation alternatives depending upon the indexes −

- **No Index** − If the table is unsorted and has no indexes, then the selection process involves scanning all the disk blocks of the table. Each block is brought into the memory and each tuple in the block is examined to see whether it satisfies the selection condition. If the condition is satisfied, it is displayed as output. This is the costliest approach since each tuple is brought into memory and each tuple is processed.

- **B+ Tree Index** − Most database systems are built upon the B+ Tree index. If the selection condition is based upon the field, which is the key of this B+ Tree index, then this index is used for retrieving results. However, processing selection statements with complex conditions may involve a larger number of disk block accesses and in some cases complete scanning of the table.
- **Hash Index** − If hash indexes are used and its key field is used in the selection condition, then retrieving tuples using the hash index becomes a simple process. A hash index uses a hash function to find the address of a bucket where the key value corresponding to the hash value is stored. In order to find a key value in the index, the hash function is executed and the bucket address is found. The key values in the bucket are searched. If a match is found, the actual tuple is fetched from the disk block into the memory.

Computation of Joins

When we want to join two tables, say P and Q, each tuple in P has to be compared with each tuple in Q to test if the join condition is satisfied. If the condition is satisfied, the corresponding tuples are concatenated, eliminating duplicate fields and appended to the result relation. Consequently, this is the most expensive operation.

The common approaches for computing joins are −

Nested-loop Approach

This is the conventional join approach. It can be illustrated through the following pseudocode (Tables P and Q, with tuples tuple_p and tuple_q and joining attribute a) −

```
For each tuple_p in P
For each tuple_q in Q
If tuple_p.a = tuple_q.a Then
   Concatenate tuple_p and tuple_q and append to Result
End If
Next tuple_q
Next tuple-p
```

Sort-merge Approach

In this approach, the two tables are individually sorted based upon the joining attribute and then the sorted tables are merged. External sorting techniques are adopted since the number of records is very high and cannot be accommodated in the memory. Once the individual tables are sorted, one page each of the sorted tables are brought to the memory, merged based upon the joining attribute and the joined tuples are written out.

Hash-join Approach

This approach comprises of two phases: partitioning phase and probing phase. In partitioning phase, the tables P and Q are broken into two sets of disjoint partitions. A common hash

function is decided upon. This hash function is used to assign tuples to partitions. In the probing phase, tuples in a partition of P are compared with the tuples of corresponding partition of Q. If they match, then they are written out.

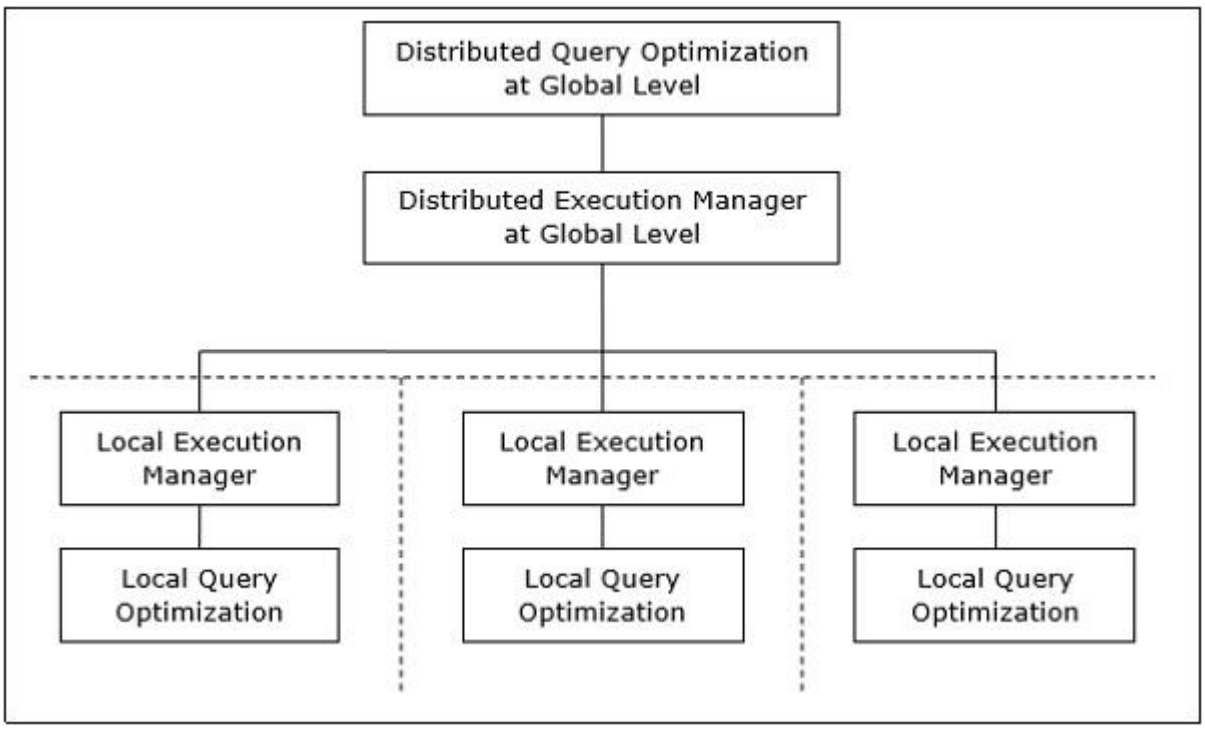
Query Processing

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram −

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.

PROJECT

PId	City	Department	Status

Suppose there is a query to retrieve details of all projects whose status is “Ongoing”.

The global query will be &inus;

$$\sigma_{status} = {\small "ongoing"}^{(PROJECT)}$$

Query in New Delhi’s server will be −

$$\sigma_{status} = {\small "ongoing"}^{({NewD}_-{PROJECT})}$$

Query in Kolkata’s server will be −

$$\sigma_{status} = {\small "ongoing"}^{({Kol}_-{PROJECT})}$$

Query in Hyderabad’s server will be −

$$\sigma_{status} = {\small "ongoing"}^{({Hyd}_-{PROJECT})}$$

In order to get the overall result, we need to union the results of the three queries as follows −

$\sigma_{status} = {\small "ongoing"}^{({NewD}_-{PROJECT})} \cup \sigma_{status} = {\small "ongoing"}^{({kol}_-{PROJECT})} \cup \sigma_{status} = {\small "ongoing"}^{({Hyd}_-{PROJECT})}$

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.

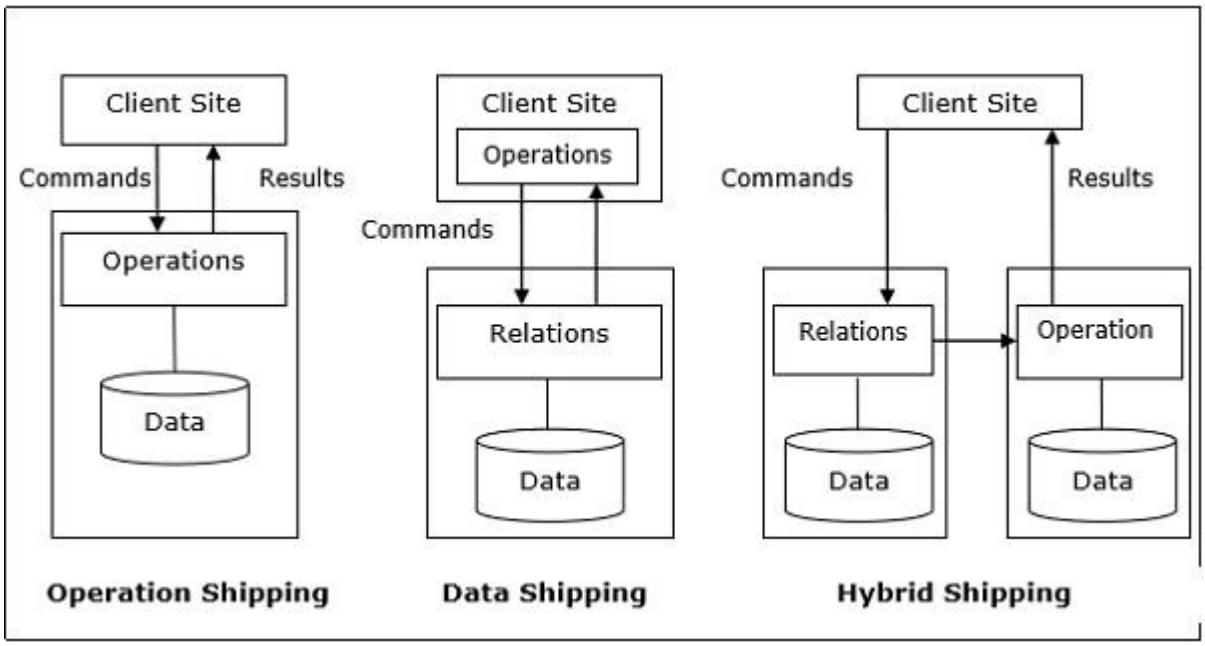
Optimal Utilization of Resources in the Distributed System

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.

Transactions

A transaction is a program including a collection of database operations, executed as a logical unit of data processing. The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data. It is an atomic process that is either performed into completion entirely or is not performed at all. A transaction involving only data retrieval without any data update is called read-only transaction.

Each high level operation can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks −

- **read_item()** − reads data item from storage to main memory.
- **modify_item()** − change value of item in the main memory.
- **write_item()** − write the modified value from main memory to storage.

Database access is restricted to read_item() and write_item() operations. Likewise, for all transactions, read and write forms the basic database operations.

Transaction Operations

The low level operations performed in a transaction are −

- **begin_transaction** − A marker that specifies start of transaction execution.

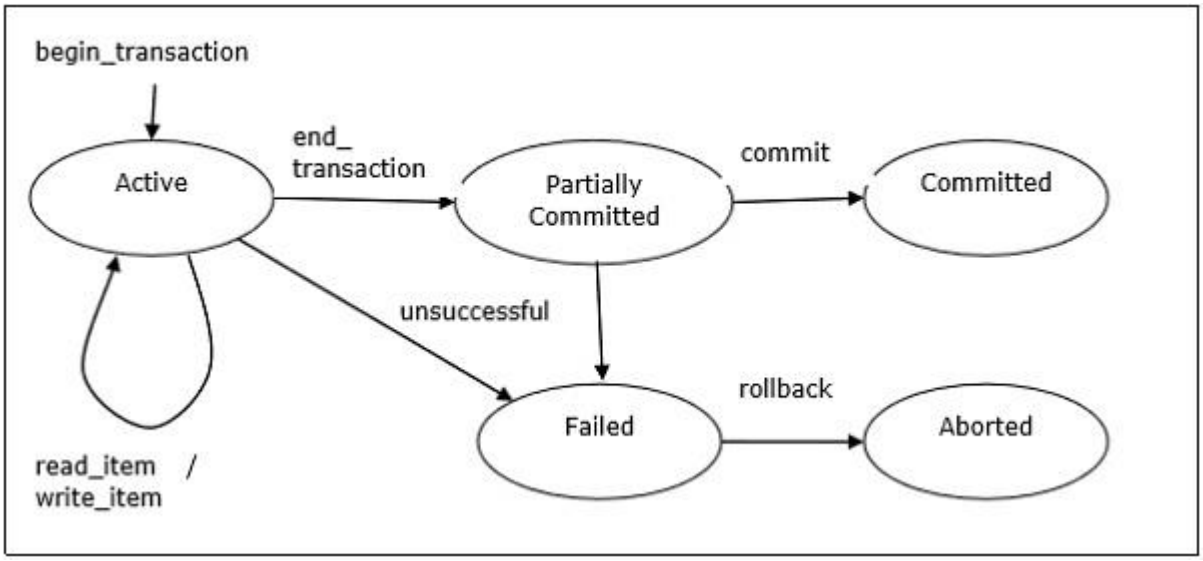
- **read_item or write_item** − Database operations that may be interleaved with main memory operations as a part of transaction.
- **end_transaction** − A marker that specifies end of transaction.
- **commit** − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
- **rollback** − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

Transaction States

A transaction may go through a subset of five states, active, partially committed, committed, failed and aborted.

- **Active** − The initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations.
- **Partially Committed** − The transaction enters this state after the last statement of the transaction has been executed.
- **Committed** − The transaction enters this state after successful completion of the transaction and system checks have issued commit signal.
- **Failed** − The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail.
- **Aborted** − This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began.

The following state transition diagram depicts the states in the transaction and the low level transaction operations that causes change in states.



Desirable Properties of Transactions

Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** − This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.
- **Consistency** − A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database.
- **Isolation** − A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.
- **Durability** − If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

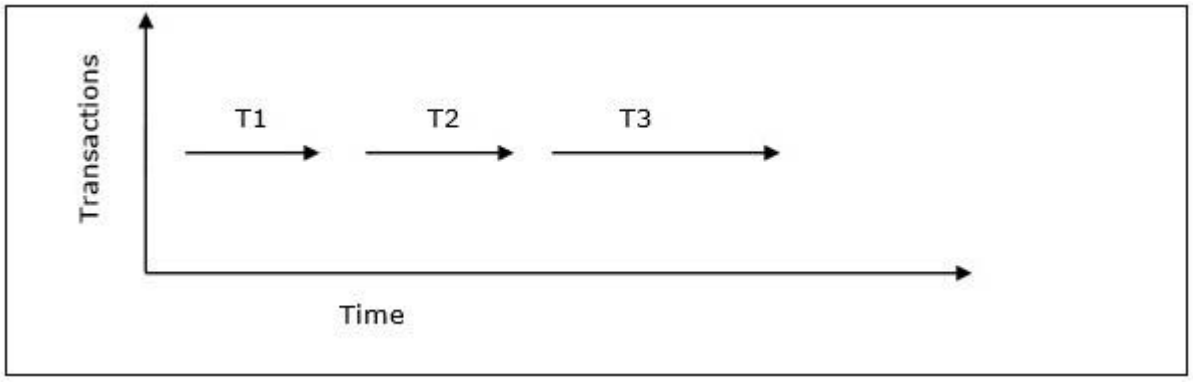
Schedules and Conflicts

In a system with a number of simultaneous transactions, a **schedule** is the total order of execution of operations. Given a schedule S comprising of n transactions, say T1, T2, T3……..Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

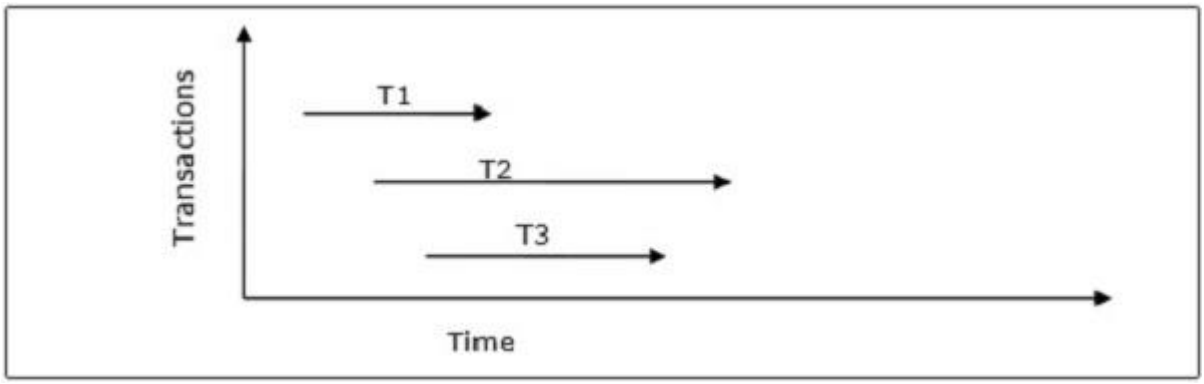
Types of Schedules

There are two types of schedules −

- **Serial Schedules** − In a serial schedule, at any point of time, only one transaction is active, i.e. there is no overlapping of transactions. This is depicted in the following graph −



- **Parallel Schedules** − In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time. This is depicted in the following graph −.



Conflicts in Schedules

In a schedule comprising of multiple transactions, a **conflict** occurs when two active transactions perform non-compatible operations. Two operations are said to be in conflict, when all of the following three conditions exists simultaneously −

- The two operations are parts of different transactions.
- Both the operations access the same data item.
- At least one of the operations is a write_item() operation, i.e. it tries to modify the data item.

Serializability

A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule.

Equivalence of Schedules

Equivalence of two schedules can be of the following types −

- **Result equivalence** − Two schedules producing identical results are said to be result equivalent.
- **View equivalence** − Two schedules that perform similar action in a similar manner are said to be view equivalent.
- **Conflict equivalence** − Two schedules are said to be conflict equivalent if both contain the same set of transactions and has the same order of conflicting pairs of operations.