Distributed Databases

Dr. Awad Khalil
Computer Science & Engineering department
Outline

- Introduction
- Distributed Database Concepts
- What Constitutes a DDB
- Transparency
- Availability and Reliability
- Scalability and Partition Tolerance
- Autonomy
- Advantages of Distributed Databases
- Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design
- Data Fragmentation and Sharding
- Data Replication and Allocation
- Types of Distributed Database Systems
- Distributed Database Architectures
- Parallel versus Distributed Architecture
- General Architecture of Pure Distributed Database
- Federated Database Schema Architecture
- An Overview of Three-Tier Client/Server Architecture
Introduction

- Distributed databases bring the advantages of distributed computing to the database domain. A distributed computing system consists of a number of processing sites or nodes that are interconnected by a computer network and that cooperate in performing certain assigned tasks.

- As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. Thus, more computing power is harnessed to solve a complex task, and the autonomous processing nodes can be managed independently while they cooperate to provide the needed functionalities to solve the problem.

- DDB technology resulted from a merger of two technologies: database technology and distributed systems technology.
Introduction

Several distributed database prototype systems were developed in the 1980s and 1990s to address the issues of data distribution, data replication, distributed query and transaction processing, distributed database metadata management, and other topics.

More recently, many new technologies have emerged that combine distributed and database technologies. These technologies and systems are being developed for dealing with the storage, analysis, and mining of the vast amounts of data that are being produced and collected, and they are referred to generally as **big data technologies**. The origins of big data technologies come from distributed systems and database systems, as well as data mining and machine learning algorithms that can process these vast amounts of data to extract needed knowledge.
Distributed Database Concepts

- We can define a distributed database (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a distributed database management system (DDBMS) as a software system that manages a distributed database while making the distribution transparent to the user.
What Constitutes a DDB

For a database to be called distributed, the following minimum conditions should be satisfied:

- Connection of database nodes over a computer network. There are multiple computers, called sites or nodes. These sites must be connected by an underlying network to transmit data and commands among sites.
- Logical interrelation of the connected databases. It is essential that the information in the various database nodes be logically related.
- Possible absence of homogeneity among connected nodes. It is not necessary that all nodes be identical in terms of data, hardware, and software.
- For an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying network are invisible to the end user.
What Constitutes a DDB

The sites may all be located in physical proximity - say, within the same building or a group of adjacent buildings - and connected via a local area network, or they may be geographically distributed over large distances and connected via a long-haul or wide area network. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.
What Constitutes a DDB

Networks may have different topologies that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site.
Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part.

In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced.

Consider the company database. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows) and stored with possible replication, as shown in the following Figure. The following types of transparencies are possible:
## Schema Diagram for the COMPANY Relational Database Schema

### EMPLOYEE

<table>
<thead>
<tr>
<th>Fname</th>
<th>Minit</th>
<th>Lname</th>
<th>Ssn</th>
<th>Bdate</th>
<th>Address</th>
<th>Sex</th>
<th>Salary</th>
<th>Super_ssn</th>
<th>Dno</th>
</tr>
</thead>
</table>

### DEPARTMENT

<table>
<thead>
<tr>
<th>Dname</th>
<th>Dnumber</th>
<th>Mgr_ssn</th>
<th>Mgr_start_date</th>
</tr>
</thead>
</table>

### DEPT_LOCATIONS

<table>
<thead>
<tr>
<th>Dnumber</th>
<th>Dlocation</th>
</tr>
</thead>
</table>

### PROJECT

<table>
<thead>
<tr>
<th>Pname</th>
<th>Pnumber</th>
<th>Plocation</th>
<th>Dnum</th>
</tr>
</thead>
</table>

### WORKS_ON

<table>
<thead>
<tr>
<th>Essn</th>
<th>Pno</th>
<th>Hours</th>
</tr>
</thead>
</table>

### DEPENDENT

<table>
<thead>
<tr>
<th>Essn</th>
<th>Dependent_name</th>
<th>Sex</th>
<th>Bdate</th>
<th>Relationship</th>
</tr>
</thead>
</table>
Transparency
Data organization transparency (also known as distribution or network transparency). This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency.

- Location transparency refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued.

- Naming transparency implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.

Replication transparency. As we show in the Figure, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
Transparency

- **Fragmentation transparency.** Two types of fragmentation are possible. *Horizontal fragmentation* distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation; this is also known as **sharding** in the newer **big data** and **cloud computing** systems. *Vertical fragmentation* distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include design transparency and execution transparency - which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.
Reliability and availability are two of the most common potential advantages cited for distributed databases. Reliability is broadly defined as the probability that a system is running (not down) at a certain time point, whereas availability is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. Errors constitute that subset of system states that causes the failure. Fault is the cause of an error.
To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components, and it processes user requests as long as data base consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Network failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.
Scalability and Partition Tolerance

Scalability determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

- **Horizontal scalability**: This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.

- **Vertical scalability**: This refers to expanding the capacity of the individual nodes in the system, such as expanding the storage capacity or the processing power of a node.
Scalability and Partition Tolerance

As the system expands its number of nodes, it is possible that the network, which connects the nodes, may have faults that cause the nodes to be partitioned into groups of nodes. The nodes within each partition are still connected by a subnetwork, but communication among the partitions is lost. The concept of **partition tolerance** states that the system should have the capacity to continue operating while the network is partitioned.
Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution.

- **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes.
- **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes.
- **Execution autonomy** refers to independence of users to act as they please.
Advantages of Distributed Databases

- **Improved ease and flexibility of application development.** Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.

- **Increased availability.** This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network.

- **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks.

- **Easier expansion via scalability.** In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems.
Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

- **Data Fragments** are the techniques that are used to break up the database into logical units, called fragments, which may be assigned for storage at the various nodes.

- **Replication** is the technique that permits certain data to be stored in more than one site to increase availability and reliability; and the process of allocating fragments - or replicas of fragments - for storage at the various nodes. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.
Data Fragmentation and Sharding

In a DDB, decisions must be made regarding which site should be used to store which portions of the database.

Before we decide on how to distribute the data, we must determine the logical units of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each whole relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure S.6, and assume there are three computer sites—one for each department in the company.

We may want to store the database information relating to each department at the computer site for that department. A technique called **horizontal fragmentation** or **sharding** can be used to partition each relation by department.
# Data Fragmentation and Sharding

## EMPLOYEE

<table>
<thead>
<tr>
<th>Name</th>
<th>Minit</th>
<th>Lname</th>
<th>Ssn</th>
<th>Bdate</th>
<th>Address</th>
<th>Sex</th>
<th>Salary</th>
<th>Super_ssn</th>
<th>Dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>B</td>
<td>Smith</td>
<td>123456789</td>
<td>1965-01-09</td>
<td>731 Fondren, Houston, TX</td>
<td>M</td>
<td>30000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Franklin</td>
<td>T</td>
<td>Wong</td>
<td>333445555</td>
<td>1955-12-08</td>
<td>638 Voss, Houston, TX</td>
<td>M</td>
<td>40000</td>
<td>888665555</td>
<td>5</td>
</tr>
<tr>
<td>Alicia</td>
<td>J</td>
<td>Zelaya</td>
<td>999887777</td>
<td>1968-01-19</td>
<td>3321 Castle, Spring, TX</td>
<td>F</td>
<td>25000</td>
<td>987654321</td>
<td>4</td>
</tr>
<tr>
<td>Jennifer</td>
<td>S</td>
<td>Wallace</td>
<td>987654321</td>
<td>1941-06-20</td>
<td>291 Berry, Bellaire, TX</td>
<td>F</td>
<td>43000</td>
<td>888665555</td>
<td>4</td>
</tr>
<tr>
<td>Ramesh</td>
<td>K</td>
<td>Narayan</td>
<td>666684444</td>
<td>1962-09-15</td>
<td>975 Fire Oak, Humble, TX</td>
<td>M</td>
<td>38000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Joyce</td>
<td>A</td>
<td>English</td>
<td>453453453</td>
<td>1972-07-31</td>
<td>5631 Rice, Houston, TX</td>
<td>F</td>
<td>25000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Ahmad</td>
<td>V</td>
<td>Jabbar</td>
<td>987987987</td>
<td>1969-03-29</td>
<td>980 Dallas, Houston, TX</td>
<td>M</td>
<td>25000</td>
<td>987654321</td>
<td>4</td>
</tr>
<tr>
<td>James</td>
<td>E</td>
<td>Borg</td>
<td>888665555</td>
<td>1937-11-10</td>
<td>450 Stone, Houston, TX</td>
<td>M</td>
<td>55000</td>
<td>NULL</td>
<td>1</td>
</tr>
</tbody>
</table>

## DEPARTMENT

<table>
<thead>
<tr>
<th>Name</th>
<th>Dnumber</th>
<th>Mgr_ssn</th>
<th>Mgr_start_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td>5</td>
<td>333445555</td>
<td>1988-05-22</td>
</tr>
<tr>
<td>Administration</td>
<td>4</td>
<td>987654321</td>
<td>1995-01-01</td>
</tr>
<tr>
<td>Headquarters</td>
<td>1</td>
<td>888665555</td>
<td>1981-06-19</td>
</tr>
</tbody>
</table>

## DEPT_LOCATIONS

<table>
<thead>
<tr>
<th>Dnumber</th>
<th>Dlocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Houston</td>
</tr>
<tr>
<td>4</td>
<td>Stafford</td>
</tr>
<tr>
<td>5</td>
<td>Bellaire</td>
</tr>
<tr>
<td>5</td>
<td>Sugarland</td>
</tr>
<tr>
<td>5</td>
<td>Houston</td>
</tr>
</tbody>
</table>

## PROJECT

<table>
<thead>
<tr>
<th>Pname</th>
<th>Pnumber</th>
<th>Plocation</th>
<th>Dnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX</td>
<td>1</td>
<td>Bellaire</td>
<td>5</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td>Sugarland</td>
<td>5</td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td>Houston</td>
<td>5</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>Stafford</td>
<td>4</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td>Houston</td>
<td>1</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td>Stafford</td>
<td>4</td>
</tr>
</tbody>
</table>

## DEPENDENT

<table>
<thead>
<tr>
<th>Essn</th>
<th>Dependent_name</th>
<th>Sex</th>
<th>Bdate</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>333445555</td>
<td>Alice</td>
<td>F</td>
<td>1986-04-05</td>
<td>Daughter</td>
</tr>
<tr>
<td>333445555</td>
<td>Theodore</td>
<td>M</td>
<td>1983-10-25</td>
<td>Son</td>
</tr>
<tr>
<td>333445555</td>
<td>Joy</td>
<td>F</td>
<td>1958-05-03</td>
<td>Spouse</td>
</tr>
<tr>
<td>987654321</td>
<td>Abner</td>
<td>M</td>
<td>1942-02-28</td>
<td>Spouse</td>
</tr>
<tr>
<td>123456789</td>
<td>Michael</td>
<td>M</td>
<td>1986-01-04</td>
<td>Son</td>
</tr>
<tr>
<td>123456789</td>
<td>Alice</td>
<td>F</td>
<td>1988-12-30</td>
<td>Daughter</td>
</tr>
<tr>
<td>123456789</td>
<td>Elizabeth</td>
<td>F</td>
<td>1967-05-05</td>
<td>Spouse</td>
</tr>
</tbody>
</table>
Horizontal Fragmentation (Sharding). A horizontal fragment or shard of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment can be specified by a condition on one or more attributes of the relation, or by some other mechanism. Often, only a single attribute is involved in the condition. For example, we may define three horizontal fragments on the EMPLOYEE relation with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1) – each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1) - each fragment contains the PROJECT tuples controlled by a particular department. Horizontal fragmentation divides a relation horizontally by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites (nodes) in the distributed system. Derived horizontal fragmentation applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. Thus, related data between the primary and the secondary relations gets fragmented in the same way.
**Vertical Fragmentation.** Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. Vertical fragmentation divides a relation "vertically" by columns. A vertical fragment of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information - Name, Bdate, Address, and Sex - and the second includes work-related information - Ssn, Salary, Super_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together since there is no common attribute between the two fragments. It is necessary to include the primary key or some unique key attribute in every vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.
Notice that each horizontal fragment on a relation R can be specified in the relational algebra by a \( \sigma \) operation. A set of horizontal fragments whose conditions \( C_1, C_2, \ldots, C_n \) include all the tuples in R—that is, every tuple in R satisfies \( (C_1 \text{ OR } C_2 \text{ OR } \ldots \text{ OR } C_n) \)—is called a complete horizontal fragmentation of R. In many cases a complete horizontal fragmentation is also disjoint; that is, no tuple in R satisfies \( (C_i \text{ AND } C_j) \) for any \( i \neq j \). Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a complete horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a \( \Pi \) operation in the relational algebra. A set of vertical fragments whose projection lists \( L_1, L_2, \ldots, L_n \) include all the attributes in R but share only the primary key attribute of R is called a complete vertical fragmentation of R. In this case the projection lists satisfy the following two conditions:

- \( L_1 \cup L_2 \cup \ldots \cup L_n = \text{ATTRS}(R) \)
- \( L_i \cap L_j = \text{PK}(R) \) for any \( i \neq j \), where \( \text{ATTRS}(R) \) is the set of attributes of R and \( \text{PK}(R) \) is the primary key of R
Mixed (Hybrid) Fragmentation. We can intermix the two types of fragmentation, yielding a mixed fragmentation. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION and OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a fragment of a relation R can be specified by a SELECT - PROJECT combination of operations nL(\text{ac}(R))$. If C = TRUE (that is, all tuples are selected) and L := \text{ATTRS}(R)$, we get a vertical fragment, and if C := TRUE and L = \text{ATTRS}(R)$, we get a horizontal fragment. Finally, if C := TRUE and L := \text{ATTRS}(R)$, we get a mixed fragment. Notice that a relation can itself be considered a fragment with C = TRUE and L = \text{ATTRS}(R)$. In the following discussion, the term fragment is used to refer to a relation or to any of the preceding types of fragments.
Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the whole database at every site in the distributed system, thus creating a fully replicated distributed database. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval (read performance) for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations (write performance) drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication.
The other extreme from full replication involves having no replication that is, each fragment is stored at exactly one site. In this case, all fragments must be disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called nonredundant allocation.

Between these two extremes, we have a wide spectrum of partial replication of the data - that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers - such as sales forces, financial planners, and claims adjustors - carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a replication schema.
The term **distributed database management** system can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the degree of local **autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**. On the other hand, if direct access by local transactions to a server is permitted, the system has some degree of **local autonomy**.
Types of Distributed Database Systems

The following Figure shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is complete autonomy but a total lack of distribution and heterogeneity (point A in the figure). We see that the degree of local autonomy provides further ground for classification into federated and multi-database systems. At one extreme of the autonomy spectrum, we have a DDBMS that looks like a centralized DBMS to the user, with zero autonomy (point B). A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS - which means that no local autonomy exists. Along the autonomy axis we encounter two types of DDBMSs called federated database system (point C) and multi-database system (point D). In such systems, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA, and hence has a very high degree of local autonomy.
Types of Distributed Database Systems

Legend:
A: Traditional centralized database systems
B: Pure distributed database systems
C: Federated database systems
D: Multidatabase or peer-to-peer database systems
The term federated database system (FDBS) is used when there is some global view or schema of the federation of databases that is shared by the applications (point C). On the other hand, a multi-database system has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (point D). Both systems are hybrids between distributed and centralized systems, and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense. Point D in the diagram may also stand for a system with full local autonomy and full heterogeneity - this could be a peer-to-peer database system. In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS (such as Computer Associates' IDMS or HP'S IMAGE/3000), and a third an object DBMS (such as Object Design's ObjectStore) or hierarchical DBMS (such as IBM's IMS); in such a case, it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server.
In this section, we first briefly point out the distinction between parallel and distributed database architectures. Although both are prevalent in industry today, there are various manifestations of the distributed architectures that are continuously evolving among large enterprises. The parallel architecture is more common in high-performance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications.
Parallel Versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

- **Shared memory** (tightly coupled) architecture. Multiple processors share secondary (disk) storage and also share primary memory.
- **Shared disk** (loosely coupled) architecture. Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network. A Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called shared-nothing architecture. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high speed interconnection network (bus or switch).
Parallel Versus Distributed Architectures

Although the shared-nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared-nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment, where heterogeneity of hardware and operating system at each node is very common. Shared-nothing architecture is also considered as an environment for parallel databases. The Figure (a) illustrates a parallel database (shared nothing), whereas Figure (b) illustrates a centralized database with distributed access and Figure (c) shows a pure distributed database.
Parallel Versus Distributed Architectures

(a) Parallel Architecture

Computer System 1

CPU

Memory

DB

Switch

Computer System 2

CPU

Memory

DB

Computer System n

CPU

Memory

DB

(b) Distributed Architecture

DB_1

Central Site (Chicago)

DB_2

Site (San Francisco)

Site (Los Angeles)

Communications Network

Site (New York)

Site (Atlanta)

(c) Hybrid Architecture

Communications Network

Site 5

Site 4

Site 3

Site 1

Site 2
In this section, we discuss both the logical and component architectural models of a DDB. In the following Figure, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency. To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency. The following Figure shows the component architecture of a DDB.
General Architecture of Pure Distributed Database

User

External View

Global Conceptual Schema (GCS)

Local Conceptual Schema (LCS)

Local Internal Schema (LIS)

Stored Data

Site 1

Sites 2 to n-1

Site n

User

External View

Local Conceptual Schema (LCS)

Local Internal Schema (LIS)

Stored Data
Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in the following Figure. In this architecture, the local schema is the conceptual schema (full database definition) of a component database, and the component schema is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The export schema represents the subset of a component schema that is available to the FDBS. The federated schema is the global schema or view, which is the result of integrating all the shareable export schemas. The external schemas define the schema for a user group or an application, as in the three-level schema architecture.

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations.
Federated Database Schema Architecture
An Overview of Three-Tier Client/Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client/server architectures. It is now more common to use a three-tier architecture rather than a two-tier architecture, particularly in Web applications. This architecture is illustrated in the following Figure.

In the three-tier client/server architecture, the following three layers exist:

- **Presentation layer {client}**. This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
An Overview of Three-Tier Client/Server Architecture
An Overview of Three-Tier Client/Server Architecture

- **Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

- **Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational, and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML when transmitted between the application server and the database server.
Other Issues Related to DDBs

- Concurrency Control and Recovery
- Transaction Management
- Query Processing and Optimization