Distributed Database Design
Objectives

- Definition of Distributed Database
- Motivation for Distributed Database
- Advantage of Distributed Database
- Disadvantages of Distributed Database
- Transactions Management in a Distributed Database environment
- Design of the Distributed Database environment
• Distributed database management system (DDBMS)
  – Distributed Database: A logically interrelated collection of shared data, physically distributed over a computer network
  – Distributed DBMS (DDBMS): Software system that permits the management of the distributed database and makes the distribution transparent to users
    • Governs storage and processing of logically related data over interconnected computer systems in which both data and processing functions are distributed among several sites
Motivation for Distributed Database

- The development of computer network promotes de-centralization
- In a company, the database organization might reflect the organizational structure, which is distributed into units. Each unit maintains its own database
- Sharing of data can be achieved by developing a distributed database system which:
  - Makes data accessible by all units
  - Stores data close to where it is most frequently used
Distributed Processing Environment

Distributed Database Design

Computer Network

Bangalore

Chennai

Mangalore

Corporate Employee Database
Distributed Database Management System
DDBMS Advantages

- Data are located near “greatest demand” site
- Faster data access
- Faster data processing
- Growth facilitation
- Improved communications
- Reduced operating costs
- User-friendly interface
- Less danger of a single-point failure
- Processor independence
DDBMS Disadvantages

- Complexity of management and control
- Security
- Lack of standards
- Increased storage requirements
- Greater difficulty in managing the data environment
- Increased training cost
Characteristics of Distributed Management Systems

• Collection of logically-related shared data
• Data split into fragments
• Fragments may be replicated
• Fragments/replicas allocated to sites
• Sites linked by a communications network
• Data at each site is under control of a DBMS
• DBMSs handle local applications autonomously
• Each DBMS participates in at least one global application
Characteristics of Distributed Management Systems

• Must perform all the functions of a centralized DBMS

• Must handle all necessary functions imposed by the distribution of data and processing

• Must perform these additional functions *transparently* to the end user
DDBMS Components

• Must include (at least) the following components:
  – *Computer workstations*
  – *Network hardware and software*
  – *Communications media*
  – Transaction processor (or, application processor, or transaction manager)
    • Software component found in each computer that requests data
  – Data processor or data manager
    • Software component residing on each computer that stores and retrieves data located at the site
Distributed Database Design

**Single-Site Processing, Single-Site Data (SPSD)**

- All processing is done on single CPU or host computer (mainframe, midrange, or PC)
- All data are stored on host computer’s local disk
- Processing cannot be done on end user’s side of the system
- Typical of most mainframe and midrange computer DBMSs
- DBMS is located on the host computer, which is accessed by dumb terminals connected to it
- Also typical of the first generation of single-user microcomputer databases
Multiple-Site Processing, Single-Site Data (MPSD)

- Multiple processes run on different computers sharing a single data repository
- MPSD scenario requires a network file server running conventional applications that are accessed through a LAN
- Many multi-user accounting applications, running under a personal computer network, fit such a description
Multiple-Site Processing, Multiple-Site Data (MPMD)

- Fully distributed database management system with support for multiple data processors and transaction processors at multiple sites
- Classified as either homogeneous or heterogeneous
- Homogeneous DDBMSs
  - Integrate only one type of centralized DBMS over a network
Homogeneous Distributed Database
Multiple-Site Processing, Multiple-Site Data (MPMD)

- **Heterogeneous DDBMSs**
  - Integrate different types of centralized DBMSs over a network

- **Fully heterogeneous DDBMS**
  - Support different DBMSs that may even support different data models (relational, hierarchical, or network) running under different computer systems, such as mainframes and microcomputers
Distributed Database Transparency Features

- Allow end user to feel like database’s only user
- Features include:
  - Distribution transparency
  - Transaction transparency
  - Failure transparency
  - Performance transparency
  - Heterogeneity transparency
Distribution Transparency

- Allows management of a physically dispersed database as though it were a centralized database
- Three levels of distribution transparency are recognized:
  - Fragmentation transparency
  - Location transparency
  - Local mapping transparency
Reference Architecture of DDBMS

Due to diversity, no accepted architecture equivalent to ANSI/SPARC 3-level architecture for DBMSs.

A possible reference architecture consists of:

- Set of global external schemas.
- Global conceptual schema (GCS).
- Fragmentation schema and allocation schema.
- Set of schemas for each local DBMS conforming to 3-level ANSI/SPARC.
- Some levels may be missing, depending on levels of transparency supported.
Reference Architecture of DDBMS

S1
- Global external schema
- Global conceptual schema
- Fragmentation schema
- Allocation schema
- Local mapping schema
- Local conceptual schema
- Local internal schema
- DB

S2

... (for S3 to Sn)

S1
- Global external schema
- Global conceptual schema
- Fragmentation schema
- Allocation schema
- Local mapping schema
- Local conceptual schema
- Local internal schema
- DB

Distributed Database Design
Reference Architecture of DDBMS

- Global Conceptual Schema is the logical description of the DB as if it were not distributed. It contains definitions of entities, relationships, constraints, security, and integrity information.
- Fragmentation and Allocation Schemas describe how data are logically partitioned, and where they are located, taking replication into account.
- Local Schemas are the logical descriptions of the local DBs.
Transaction Transparency

- Ensures database transactions will maintain distributed database’s integrity and consistency
Distributed Requests and Distributed Transactions

- Distributed transaction
  - Can update or request data from several different remote sites on a network
- Remote request
  - Lets a single SQL statement access data to be processed by a single remote database processor
- Remote transaction
  - Accesses data at a single remote site
Distributed Queries and Distributed Transactions

• Distributed Queries
  – Lets a single SQL statement reference data located at several different local or remote DB sites

• Distributed transaction
  – Allows a transaction to reference several different (local or remote) DB sites
Distributed Query

SELECT ename, dname
    FROM Hanu.emp e,
    Hanu.dept@sales.in.auto.com d
WHERE e.deptno = d.deptno;
Distributed Transaction

BEGIN
UPDATE Hanu.dept@sales.in.auto.com
SET loc = 'Gulbarga'
WHERE deptno = 10;
UPDATE Hanu.emp
SET deptno = 11
WHERE deptno = 10;
COMMIT;
Distributed Concurrency Control

- Multisite, multiple-process operations are much more likely to create data inconsistencies and deadlocked transactions than are single-site systems.
Issues in Distributed Database Design

Three key issues we have to consider:

- **Data Allocation**: where are data placed?
  
  Data should be stored at site with "optimal" distribution.

- **Fragmentation**: relation may be divided into a number of sub-relations (called fragments), which are stored in different sites.

- **Replication**: copy of fragment may be maintained at several sites.
Issues in Distributed Database Design

• Definition and allocation of fragments carried out strategically to achieve:
  – Locality of Reference
  – Improved Reliability and Availability
  – Improved Performance
  – Balanced Storage Capacities and Costs
  – Minimal Communication Costs.

• Involves analyzing most important transactions, based on quantitative/qualitative information.
Data Allocation

- Four strategies regarding placement of data:
  - Centralized
  - Partitioned (or Fragmented)
  - Complete Replication
  - Selective Replication
Data Allocation

- Centralized: Consists of single database stored at one site with users distributed across the network (This is not a DDB but distributed processing!)
- Partitioned: Database partitioned into disjoint fragments, each fragment assigned to one site.
- Complete Replication: Consists of maintaining complete copy of database at each site.
- Selective Replication: Combination of partitioning, replication, and centralization.
Why Fragment?

- **Usage**
  - Applications work with partition rather than entire relations
- **Efficiency**
  - Data is stored close to where it is most frequently used
  - Data that is not needed by local applications is not stored
- **Parallelism**
  - With fragments as unit of distribution, transaction can be divided into several sub-queries that operate on fragments
- **Security**
  - Data not required by local applications is not stored and so not available to unauthorized users
Design Consideration for Fragmentations

- Quantitative information may include:
  - frequency with which a transaction is run;
  - site from which a transaction is run;
  - performance criteria for transactions.
- Qualitative information may include transactions that are executed such as:
  - type of access (read or write);
  - predicates of read operations.
Fragmentation

- A relation R is divided into fragments r1, r2, …rn, which contain enough information to allow reconstruction of R
- Example:
  We have a relation Sells (pub, address, price, type)
  Type is “small” or “large”. We can split Sells into two different fragments:
  - Sellssmall= σtype = “small”(Sells)
  - SellsLage= σtype = “large”(Sells)
## Comparison of Strategies for Data Distribution

<table>
<thead>
<tr>
<th></th>
<th>Locality of reference</th>
<th>Reliability and availability</th>
<th>Performance</th>
<th>Storage costs</th>
<th>Communication costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>Lowest</td>
<td>Lowest</td>
<td>Unsatisfactory</td>
<td>Lowest</td>
<td>Highest</td>
</tr>
<tr>
<td>Fragmented</td>
<td>High&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Low for item; high for system</td>
<td>Satisfactory&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Lowest</td>
<td>Low&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Complete replication</td>
<td>Highest</td>
<td>Highest</td>
<td>Best for read</td>
<td>Highest</td>
<td>High for update; low for read</td>
</tr>
<tr>
<td>Selective replication</td>
<td>High&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Low for item; high for system</td>
<td>Satisfactory&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Average</td>
<td>Low&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>a</sup> Indicates subject to good design.
Types of Fragmentation

• Four types of fragmentation:
  – Horizontal
  – Vertical
  – Mixed
  – Derived

• Other possibility is no fragmentation:
  – If relation is small and not updated frequently, may be better not to fragment relation.
Horizontal and Vertical Fragmentation

Horizontal

Vertical
Horizontal Fragmentation

- Each fragment consists of a subset of the tuples of a relation R.
- Defined using Selection operation of relational algebra: $\sigma p(R)$
- Example:
  - Relation: Sells(pub, address, price, type)
  - Fragments:
    - SellsBitter $= \sigma $type = “bitter”(Sells)
    - SellsLager $= \sigma $type = “lager”(Sells)
Horizontal Fragmentation

- This strategy is determined by looking at predicates used by transactions.
- Involves finding set of *minimal* (*complete* and *relevant*) predicates.
- Set of predicates is *complete*, if and only if, any two tuples in same fragment are referenced with same probability by any application.
- Predicate is *relevant* if there is at least one application that accesses fragments differently
Vertical Fragmentation

- Each fragment consists of a subset of attributes of a relation \( R \).
- Defined using projection operation of relational algebra: \( \Pi_{a_1, \ldots, a_n}(R) \)
- Determined by establishing affinity of one attribute to another.
- Example:
  - Relation: \( \text{Bars} \) (name, address, licence, employees, owner)
  - Fragments:
    - \( \Pi_{\text{name, address, licence}}(\text{Bars}) \)
    - \( \Pi_{\text{name, address, employees, owner}}(\text{Bars}) \)
Mixed Fragmentation
Example - Mixed Fragmentation

- $S_1 = \Pi_{\text{staffNo}, \text{position}, \text{sex}, \text{DOB}, \text{salary}}(\text{Staff})$
- $S_2 = \Pi_{\text{staffNo}, \text{fName}, \text{lName}, \text{branchNo}}(\text{Staff})$
- $S_{21} = \sigma \text{branchNo} = \text{‘B003’}(S_2)$
- $S_{22} = \sigma \text{branchNo} = \text{‘B005’}(S_2)$
- $S_{23} = \sigma \text{branchNo} = \text{‘B007’}(S_2)$
Derived Horizontal Fragmentation

- A horizontal fragment that is based on horizontal fragmentation of a parent relation.
- Ensures that fragments that are frequently joined together are at same site.
- Defined using **Semijoin** operation of relational algebra:
- \( Ri = R \triangleright F Si, 1 \leq i \leq w \)
Derived Horizontal Fragmentation

- S3 = \( \sigma \) branchNo='B003'(Staff)
- S4 = \( \sigma \) branchNo='B005'(Staff)
- S5 = \( \sigma \) branchNo='B007'(Staff)
- Could use derived fragmentation for Property:
  \( Pi = PropertyForRent >branchNo Si, 3 \leq i \leq 5 \)
Derived Horizontal Fragmentation

- If relation contains more than one foreign key, need to select one as parent.
- Choice can be based on fragmentation used most frequently or fragmentation with better join characteristics.
Correctness of Fragmentation

- In defining fragments we have to be very careful.
- Three correctness rules:
  - Completeness
  - Reconstruction
  - Disjointness.
Completeness of Fragmentation

- Completeness: If relation R is decomposed into fragments r1, r2, …rn, each data item that can be found in R must appear in at least one fragment. This ensures no loss of data during fragmentation.
Reconstruction of Fragmentation

- Reconstruction: we must be able to reconstruct the entire R from fragments.
- For horizontal fragmentation is union operation.
  - \[ R = r_1 \cup r_2 \cup ... \cup r_n, \]
- For vertical fragmentation is natural join operation.
  - \[ R = r_1 \bowtie r_2 \bowtie ... \bowtie r_n, \]
- To ensure reconstruction we have to include primary key attributes in all fragments.
Disjointness of Fragmentation

- Disjointness: if data item x appears in fragment ri, then it should not appear in any other fragment.
  - Exception: vertical fragmentation, where primary key attributes must be repeated to allow reconstruction.
- For horizontal fragmentation, data item is a tuple
- For vertical fragmentation, data item is an attribute.
Correctness of Horizontal Fragment

- Relation: Sells(pub, address, price, type) type={Bitter, Lager}
- Fragments:
  - SellsBitter = \( \sigma \) type = “bitter”(Sells)
  - SellsLager = \( \sigma \) type = “lager”(Sells)
- Correctness rules
  - Completeness: Each tuple in the relation appears either in SellsBitter or in SellsLager
  - Reconstruction: The Sells relation can be reconstructed from the fragments Sells = SellsBitter \( \cup \) SellsLager
  - Disjointness: The two fragments are disjoint, there can be no beer that is both “Lager” and “Bitter”
Correctness of Vertical Fragment

- Relation: Bars(name, address, licence, employees, owner)
- Fragments:
  - \( r_1 = \Pi \text{name, address, licence} (\text{Bars}) \)
  - \( r_2 = \Pi \text{name, address, employees, owner}(\text{Bars}) \)
- Correctness rules
  - **Completeness**: Each attribute in the Bars relation appears either in \( r_1 \) or in \( r_2 \)
  - **Reconstruction**: The Bars relation can be reconstructed from the fragments
    - Bars = \( r_1 \bowtie r_2 \)
  - **Disjointness**: The two fragments are disjoint, except for the primary key, name, which is necessary for reconstruction
Transparency in Distributed databases

- Distribution Transparency
- Transaction Transparency
- Performance Transparency
- DBMS Transparency
Distribution Transparency

- The user has to perceive the DDB as a single, logical entity
- Fragmentation Transparency: the user does not need to know that data is fragmented
- Location Transparency: the user does not need to know the location of data items
- Replication Transparency: the user is unaware of replication of data.
- Naming transparency: items in a database must have a unique name, but users don’t need to worry about it.
Naming Transparency

• Each item in a DDB must have a unique name.
• DDBMS must ensure that no two sites create a database object with same name.
• Solution 1: create central name server.
• Disadvantages:
  – loss of some local autonomy;
  – central site may become a bottleneck;
  – low availability; if the central site fails, remaining sites cannot create any new objects.
Naming Transparency

- Solution 2: prefix object with identifier of site that created it
- Example: Beer created at site S1 might be named S1.Beer
- Disadvantage: loss of distribution transparency
Transaction Transparency

- Ensures that all distributed transactions maintain distributed database’s integrity and consistency.
- Distributed transaction accesses data stored at more than one location.
- Each transaction is divided into number of sub transactions, one for each site that has to be accessed.
- DDBMS must ensure the indivisibility of both the global transaction and each sub-transactions.
- Must ensure both concurrency transparency, and failure transparency
Concurrency Transparency

- logically consistent with results obtained if transactions executed one at a time, in some arbitrary serial order. Same fundamental principles as for centralized DBMS
- DDBMS must ensure both global and local transactions do not interfere with each other
- Similarly, DDBMS must ensure consistency of all sub transactions of global transaction.
- Techniques for concurrency control. Usually different from the ones for DBMS.
Concurrency Transparency

- Replication makes concurrency more complex. If a copy of a replicated data item is updated, update must be propagated to all copies.
- Could propagate changes as part of original transaction, making it an atomic operation.
- However, if one site holding copy is not reachable, then transaction is delayed until site is reachable.
Concurrency Transparency

• Could limit update propagation to only those sites currently available. Remaining sites updated when they become available again.
• Could allow updates to copies to happen asynchronously, sometime after the original update.
• Delay in regaining consistency may range from a few seconds to several hours.
Failure Transparency

- DDBMS must ensure atomicity and durability of global transaction.
- Means ensuring that sub-transactions of global transaction either all commit or all abort.
- Thus, DDBMS must synchronize global transaction to ensure that all sub-transactions have completed successfully before recording a final COMMIT for global transaction.
- Must do this in presence of site and network failures.
Performance Transparency

- DDBMS must perform as if it were a centralized DBMS:
- DDBMS should not suffer any performance degradation due to distributed architecture.
- DDBMS should determine most cost-effective strategy to execute a request.
Performance Transparency

• Distributed Query Processor (DQP) maps data request into ordered sequence of operations on local databases.
• It must consider fragmentation, replication, and allocation schemas.
• DQP has to decide:
  – which fragment to access;
  – which copy of a fragment to use;
  – which location to use.
Performance Transparency

- DQP produces execution strategy optimized with respect to some cost function.
- Typically, costs associated with a distributed request include:
  - I/O cost;
  - CPU cost;
  - Communication cost.
Performance Transparency - Example

- Property(Pno, City) 10000 records in London
- Renter(Rno, Max_Price) 100000 records in Glasgow
- Viewing(Pno, Rno) 1000000 records in London

```sql
SELECT p.pno
FROM property p INNER JOIN (renter r INNER JOIN viewing v ON r.rno = v.rno)
ON p.pno = v.pno
WHERE p.city='Aberdeen' AND r.max_price > 200000;
```
Performance Transparency - Example

- Assume: Each tuple in each relation is 100 characters long
- 10 renters with maximum price greater than £200,000.
- 100,000 viewings for properties in Aberdeen.
- Computation time negligible compared to communication time.
### Performance Transparency - Example

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Move Client relation to London and process query there</td>
<td>16.7 minutes</td>
</tr>
<tr>
<td>(2) Move Property and Viewing relations to Glasgow and process query there</td>
<td>28 hours</td>
</tr>
<tr>
<td>(3) Join Property and Viewing relations at London, select tuples for Aberdeen properties, and for each of these in turn, check at Glasgow to determine if associated ( \text{maxPrice} &gt; £200,000 )</td>
<td>2.3 days</td>
</tr>
<tr>
<td>(4) Select clients with ( \text{maxPrice} &gt; £200,000 ) at Glasgow and for each one found, check at London for a viewing involving that client and an Aberdeen property</td>
<td>20 seconds</td>
</tr>
<tr>
<td>(5) Join Property and Viewing relations at London, select Aberdeen properties, and project result over propertyNo and clientNo and move this result to Glasgow for matching with ( \text{maxPrice} &gt; £200,000 )</td>
<td>16.7 minutes</td>
</tr>
<tr>
<td>(6) Select clients with ( \text{maxPrice} &gt; £200,000 ) at Glasgow and move the result to London for matching with Aberdeen properties</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Distribution Transaction Management

- DDBMS must ensure:
  - synchronization of sub-transactions with other local transactions executing concurrently at a site;
  - synchronization of sub-transactions with global transactions running simultaneously at same of different sites.
- Global transaction manager (transaction coordinator) at each site, to coordinate global and local transactions initiated at that site.
Distribution Transaction Management

- Techniques for Distributed Concurrency Control must ensure distributed serializability
- Locking protocols (2PL protocol)
- Timestamping methods (extend the definition of timestamp so that it includes a site identifier)
Two-Phase Commit Protocol

• Distributed databases make it possible for a transaction to access data at several sites
• Final COMMIT must not be issued until all sites have committed their parts of the transaction
• Two-phase commit protocol requires each individual DP’s transaction log entry be written before the database fragment is actually updated
Two-Phase Commit Protocol

- All participating nodes in a distributed transaction should perform the same action:
- They should either all commit or all perform a rollback of the transaction. The database
- Automatically controls and monitors the commit or rollback of a distributed transaction and maintains the integrity of the global database (the collection of databases participating in the transaction) using the two-phase commit mechanism
- The commit mechanism has the following distinct phases
  - Prepare Phase
  - Commit Phase
  - Forget Phase
Two-phase commit

• Commit occurs in two phases
  – Voting phase
  – Actual commit
• Commit controlled by TP system
  – Distributed Transaction Coordinator (DTC)
Two-phase commit – Prepare Phase

- The initiating node, called the **global coordinator**, asks participating nodes other than the commit point site to promise to commit or roll back the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.
Two-phase commit – Commit Phase

- Participants must write transaction temporarily to durable storage
- If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.
- The global coordinator forgets about the transaction in Forget Phase.
Two-phase commit – Other possibility

• Three possibilities
  – All participants reply positive within time-out interval
    • Commit transaction
  – One or more participants reply negative
    • Abort transaction
  – One or more participants do not reply within time-out interval
    • Abort transaction
Two-phase Commit – Actual commit

• Commit transaction
  – DTC sends Commit OK message to all participants
  – All participants commit
    • Write from temporary durable storage to permanent durable storage
Two-phase Commit – Abort commit

- Abort transaction
  - When at least one participant not ready to commit or timeout
  - DTC sends Abort message to all participants
    - All participants rollback
    - Removed from temporary durable storage
Distributed Lock Management

- Normal locking strategies hold
  - Locking done using local lock manager
  - Local deadlocks can be prevented and/or resolved
  - Distributed deadlocks can happen
    - No reliable low cost algorithms for deadlock avoidance and prevention exist today
Query Optimization

- Objective of query optimization routine is to minimize total cost associated with the execution of a request
- Costs associated with a request are a function of the:
  - Access time (I/O) cost
  - Communication cost
  - CPU time cost
Query Optimization

- Must provide distribution transparency as well as \textit{replica} transparency
- Replica transparency:
  - DDBMS’s ability to hide the existence of multiple copies of data from the user
- Query optimization techniques:
  - Manual or automatic
  - Static or dynamic
  - Statistically based or rule-based algorithms
C. J. Date’s Twelve Commandments for Distributed Databases

• Fundamental Principle: To the user, a distributed system should look exactly like a non-distributed system.
1. Local site independence
2. Central site independence
3. Failure independence
4. Location transparency
5. Fragmentation transparency
6. Replication transparency
7. Distributed query processing
8. Distributed transaction processing
9. Hardware independence
10. Operating system independence
11. Network independence
12. Database independence
Summary

• Distributed database stores logically related data in two or more physically independent sites connected via a computer network
• Database is divided into fragments
• Distributed databases require distributed processing
• Main components of a DDBMS are the transaction processor and the data processor
Summary

- Current database systems can be classified by extent to which they support processing and data distribution
- DDBMS characteristics are best described as a set of transparencies
- A transaction is formed by one or more database requests
- A database can be replicated over several different sites on a computer network
Reference

- Oracle® Database Administrator's Guide 10g Release 2 (10.2)
- David Bell, Jane Grimson – Distributed Database Systems, Addison-Wesley
Thank You!