Distributed Databases

Based on

Silberschatz, Korth and Sudarhan
Distributed Databases

- Heterogeneous and Homogeneous Databases
- Distributed Data Storage
- Distributed Transactions
- Commit Protocols
- Concurrency Control in Distributed Databases
- Availability
- Distributed Query Processing
- Heterogeneous Distributed Databases
- Directory Systems
A distributed database system consists of loosely coupled sites that share no physical component.

Database systems that run on each site are independent of each other.

Transactions may access data at one or more sites.
Homogeneous Distributed Databases

- In a homogeneous distributed database:
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
Heterogeneous Distributed Databases

- In a heterogeneous distributed database:
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing
Distributed Data Storage

- Assume relational data model

- Replication
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance

- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites

- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments. The system maintains several identical replicas of each such fragment.
A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.

**Full replication** of a relation is the case where the relation is stored at all sites.

**Fully redundant** databases are those in which every site contains a copy of the entire database.
Advantages of Replication:

- **Availability**: failure of site containing relation $r$ does not result in unavailability of $r$ if replicas exist.

- **Parallelism**: queries on $r$ may be processed by several nodes in parallel.

- **Reduced data transfer**: relation $r$ is available locally at each site containing a replica of $r$. 

Based on Silberschatz, Korth and Sudarhan
Data Replication (Cont.)

Disadvantages of Replication:

- **Increased cost of updates**: each replica of relation $r$ must be updated. (Update projection)

- **Increased complexity of concurrency control**: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
  
  - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy
Data Fragmentation

- Division of relation $r$ into fragments $r_1, r_2, \ldots, r_n$ which contain sufficient information to reconstruct relation $r$.

- **Horizontal fragmentation**: each tuple of $r$ is assigned to one or more fragments.
Data Fragmentation (cont’)

- **Vertical fragmentation**: the schema for relation $r$ is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.
Horizontal Data Fragmentation (example)

- Assume a relation named “account” with following schema:

  \[(branch-name, account-number, balance)\]
## Horizontal Fragmentation of *account* Relation

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillside</td>
<td>A-305</td>
<td>500</td>
</tr>
<tr>
<td>Hillside</td>
<td>A-226</td>
<td>336</td>
</tr>
<tr>
<td>Hillside</td>
<td>A-155</td>
<td>62</td>
</tr>
</tbody>
</table>

\[
account_1 = \sigma_{branch-name="Hillside"}(account)
\]

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valleyview</td>
<td>A-177</td>
<td>205</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-402</td>
<td>10000</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-408</td>
<td>1123</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-639</td>
<td>750</td>
</tr>
</tbody>
</table>

\[
account_2 = \sigma_{branch-name="Valleyview"}(account)
\]
Horizontal Data Fragmentation

- Note that the original relation $r$ can be reconstructed by taking the union of all fragments:

$$r = r_1 \cup r_2 \cup r_3 \cup \ldots \cup r_n$$
Vertical Data Fragmentation (example)

- Assume a relation named “employee-info” with following schema:

  *(branch-name, customer-name, account-number, balance)*
## Vertical Fragmentation of `customer-info` Relation

<table>
<thead>
<tr>
<th>branch-name</th>
<th>customer-name</th>
<th>tuple-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillside</td>
<td>Lowman</td>
<td>1</td>
</tr>
<tr>
<td>Hillside</td>
<td>Camp</td>
<td>2</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Camp</td>
<td>3</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Kahn</td>
<td>4</td>
</tr>
<tr>
<td>Hillside</td>
<td>Kahn</td>
<td>5</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Kahn</td>
<td>6</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Green</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{deposit}_1 = \Pi_{\text{branch-name, customer-name, tuple-id}}(\text{customer-info})
\]

<table>
<thead>
<tr>
<th>account number</th>
<th>balance</th>
<th>tuple-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-305</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>A-226</td>
<td>336</td>
<td>2</td>
</tr>
<tr>
<td>A-177</td>
<td>205</td>
<td>3</td>
</tr>
<tr>
<td>A-402</td>
<td>10000</td>
<td>4</td>
</tr>
<tr>
<td>A-155</td>
<td>62</td>
<td>5</td>
</tr>
<tr>
<td>A-408</td>
<td>1123</td>
<td>6</td>
</tr>
<tr>
<td>A-639</td>
<td>750</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{deposit}_2 = \Pi_{\text{account-number, balance, tuple-id}}(\text{customer-info})
\]
Vertical Data Fragmentation

Note that the original relation \( r \) can be reconstructed by performing a natural join of all fragments:

\[
r = r_1 \text{ join } r_2 \text{ join } r_3 \text{ join } \ldots \text{ join } r_n
\]
Advantages of Fragmentation

- **Horizontal:**
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
Advantages of Fragmentation

- **Vertical:**
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
Data Transparency

- **Data transparency**: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system.

- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

Based on Silberschatz, Korth and Sudarhan
Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The **cost of a data transmission** over the network.
  - The **potential gain in performance** from having several sites process parts of the query in parallel.
Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation \( r \) from its fragments.
  - Replace relation \( r \) by the expression to construct relation \( r \) from its fragments.

- Consider the horizontal fragmentation of the \( account \) relation into:

  \[
  \text{account}_1 = \sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account})
  \]
  \[
  \text{account}_2 = \sigma_{\text{branch-name} = \text{"Valleyview"}}(\text{account})
  \]

The query \( \sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account}) \) becomes:

\[
\sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account}_1 \cup \text{account}_2)
\]

which is optimized into

\[
\sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account}_1) \cup \sigma_{\text{branch-name} = \text{"Hillside"}}(\text{account}_2)
\]
Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented:

  \[ \text{account} \Join \text{depositor} \Join \text{branch} \]

- \text{account} is stored at site \( S_1 \)
- \text{depositor} at \( S_2 \)
- \text{branch} at \( S_3 \)
- For a query issued at site \( S_i \), the system needs to produce the result at site \( S_i \)
Possible Query Processing Strategies

- Ship copies of all three relations to site $S_1$ and choose a strategy for processing the entire query locally at site $S_1$.

- Ship a copy of the account relation to site $S_2$ and compute $temp_1 = account \bowtie depositor$ at $S_2$. Ship $temp_1$ from $S_2$ to $S_3$, and compute $temp_2 = temp_1 \bowtie branch$ at $S_3$. Ship the result $temp_2$ to $S_1$.

- Devise similar strategies, exchanging the roles $S_1$, $S_2$, $S_3$

- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site
Join Strategies that Exploit Parallelism

- Consider \( r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \) where relation \( r_i \) is stored at site \( S_i \). The result must be presented at site \( S_1 \).
- \( r_1 \) is shipped to \( S_2 \) and \( r_1 \bowtie r_2 \) is computed at \( S_2 \):
  - simultaneously \( r_3 \) is shipped to \( S_4 \) and \( r_3 \bowtie r_4 \) is computed at \( S_4 \)
- \( S_2 \) ships tuples of \( (r_1 \bowtie r_2) \) to \( S_1 \) as they produced;
- \( S_4 \) ships tuples of \( (r_3 \bowtie r_4) \) to \( S_1 \)
- Once tuples of \( (r_1 \bowtie r_2) \) and \( (r_3 \bowtie r_4) \) arrive at \( S_1 \) \( (r_1 \bowtie r_2) \) \( (r_3 \bowtie r_4) \) is computed in parallel with the computation of \( (r_1 \bowtie r_2) \) at \( S_2 \) and the computation of \( (r_3 \bowtie r_4) \) at \( S_4 \).
Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms.
- Data models may differ (hierarchical, relational, etc.).
- Transaction commit protocols may be incompatible.
- Concurrency control may be based on different techniques (locking, timestamping, etc.).
- System-level details almost certainly are totally incompatible.
- A **multidatabase system** is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases.
  - Creates an illusion of logical database integration without any physical database integration.
Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - Technical difficulties and cost of conversion
    - Organizational/political difficulties
      - Organizations do not want to give up control on their data
      - Local databases wish to retain a great deal of autonomy
Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g. data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- Agreement on units of measure
- Variations in names
  - E.g. Köln vs Cologne, Mumbai vs Bombay
Query Processing

Several issues in query processing in a heterogeneous database

Schema translation
- Write a \textbf{wrapper} for each data source to translate data to a global schema
- Wrappers must also translate updates on global schema to updates on local schema

Limited query capabilities
- Some data sources allow only restricted forms of selections
  - E.g. web forms, flat file data sources
- Queries have to be broken up and processed partly at the source and partly at a different site

Removal of duplicate information when sites have overlapping information
- Decide which sites to execute query

Global query optimization

Based on Silberschatz, Korth and Sudarhan
Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.
Transaction System Architecture

Based on Silberschatz, Korth and Sudarhan
System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
- Network partition
  - A network is said to be partitioned when it has been split into two or more subsystems that lack any connection between them
    - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Based on Silberschatz, Korth and Sudarhan
Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The two-phase commit (2 PC) protocol is widely used
- The three-phase commit (3 PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.
Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed.
- Let $T$ be a transaction initiated at site $S_i$, and let the transaction coordinator at $S_i$ be $C_i$. 
Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction \( T_i \)
  - \( C_i \) adds the records \(<\text{prepare } T>\) to the log and forces log to stable storage
  - sends *prepare* \( T \) messages to all sites at which \( T \) executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record \(<\text{no } T>\) to the log and send *abort* \( T \) message to \( C_i \)
  - if the transaction can be committed, then:
    - add the record \(<\text{ready } T>\) to the log
    - force *all records* for \( T \) to stable storage
    - send *ready* \( T \) message to \( C_i \)
Phase 2: Recording the Decision

- $T$ can be committed of $C_i$ received a ready $T$ message from all the participating sites: otherwise $T$ must be aborted.
- Coordinator adds a decision record, $<\text{commit } T>$ or $<\text{abort } T>$, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.
Handling of Failures - Site Failure

When site $S_i$ recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain `<commit $T$>` record: site executes **redo** ($T$)
- Log contains `<abort $T$>` record: site executes **undo** ($T$)
- Log contains `<ready $T$>` record: site must consult $C_i$ to determine the fate of $T$.
  - If $T$ committed, **redo** ($T$)
  - If $T$ aborted, **undo** ($T$)
- The log contains no control records concerning $T$ replies that $S_k$ failed before responding to the **prepare $T$** message from $C_i$
  - since the failure of $S_k$ precludes the sending of such a response $C_i$ must abort $T$
  - $S_k$ must execute **undo** ($T$)
Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for $T$ is executing then participating sites must decide on $T$'s fate:
  1. If an active site contains a `<commit $T$>` record in its log, then $T$ must be committed.
  2. If an active site contains an `<abort $T$>` record in its log, then $T$ must be aborted.
  3. If some active participating site does not contain a `<ready $T$>` record in its log, then the failed coordinator $C_i$ cannot have decided to commit $T$. Can therefore abort $T$.
  4. If none of the above cases holds, then all active sites must have a `<ready $T$>` record in their logs, but no additional control records (such as `<abort $T$>` or `<commit $T$>`). In this case active sites must wait for $C_i$ to recover, to find decision.

Blocking problem: active sites may have to wait for failed coordinator to recover.
Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later
Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say $S_i$.
- When a transaction needs to lock a data item, it sends a lock request to $S_i$ and lock manager determines whether the lock can be granted immediately.
  - If yes, lock manager sends a message to the site which initiated the request.
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site.
Single-Lock-Manager Approach (Cont.)

- The transaction can read the data item from any one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure.
Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock detection
    - More on this later
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus
Primary Copy

- Choose one replica of data item to be the **primary copy**.
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item $Q$, it requests a lock at the primary site of $Q$.
  - Implicitly gets lock on all replicas of the data item
- **Benefit**
  - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation.
- **Drawback**
  - If the primary site of $Q$ fails, $Q$ is inaccessible even though other sites containing a replica may be accessible.
Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.

- When a transaction wishes to lock an unreplicated data item $Q$ residing at site $S_i$, a message is sent to $S_i$‘s lock manager.
  - If $Q$ is locked in an incompatible mode, then the request is delayed until it can be granted.
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.
Majority Protocol (Cont.)

- In case of replicated data
  - If \( Q \) is replicated at \( n \) sites, then a lock request message must be sent to more than half of the \( n \) sites at which \( Q \) is stored.
  - The transaction does not operate on \( Q \) until it has obtained a lock on a majority of the replicas of \( Q \).
  - When writing the data item, transaction performs writes on all replicas.

- Benefit
  - Can be used even when some sites are unavailable

- Drawback
  - Requires \( 2(n/2 + 1) \) messages for handling lock requests, and \( (n/2 + 1) \) messages for handling unlock requests.
  - Potential for deadlock even with single item - e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data.
Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.

- **Shared locks.** When a transaction needs to lock data item $Q$, it simply requests a lock on $Q$ from the lock manager at one site containing a replica of $Q$.

- **Exclusive locks.** When transaction needs to lock data item $Q$, it requests a lock on $Q$ from the lock manager at all sites containing a replica of $Q$.

- Advantage - imposes less overhead on **read** operations.

- Disadvantage - additional overhead on writes
Deadlock Handling

Consider the following two transactions and history, with item X and transaction $T_1$ at site 1, and item Y and transaction $T_2$ at site 2:

$T_1$: write (X)  
      write (Y)  

$T_2$: write (Y)  
      write (X)  

X-lock on X  
write (X)  

X-lock on Y  
write (Y)  
wait for X-lock on X  

Wait for X-lock on Y  

Result: deadlock which cannot be detected locally at either site
Centralized Approach

- A global wait-for graph is constructed and maintained in a single site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm.

- the global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.
  - a number of changes have occurred in a local wait-for graph.
  - the coordinator needs to invoke cycle-detection.

- If the coordinator finds a cycle, it selects a victim and notifies all sites. The sites roll back the victim transaction.
Local and Global Wait-For Graphs

Local

Global

Based on Silberschatz, Korth and Sudarhan
Availability

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- Robustness: ability of system to function despite failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery/reintegration when a site or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure