**Distributed Databases**

Part 1

Advanced Topics in Database Management (INFSCI 1022)
Distributed Databases (TELCOM 2326)


Vladimir Zadorozhny, GIST, University of Pittsburgh

Distributed Database System

- 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
- 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: system maintains several identical replicas of each such fragment.
Data Replication

- 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.

Data Replication (Cont.)

- 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 $r$.

- Disadvantages of Replication
  - Increased cost of updates: each replica of relation $r$ must be updated.
  - 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
- **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.
- Example: relation account with following schema
- \( \text{Account} = (\text{branch}\_\text{name}, \text{account}\_\text{number}, \text{balance}) \)

### 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>

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

<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>

\[ \text{account}_2 = \sigma_{\text{branch}\_\text{name}=\text{"Valleyview"}}(\text{account}) \]
Vertical Fragmentation of `employee_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>

\[
deposit_1 = \Pi_{branch\_name, customer\_name, tuple\_id}(employee\_info)
\]

<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>

\[
deposit_2 = \Pi_{account\_number, balance, tuple\_id}(employee\_info)
\]

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
- 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
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.
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

Naming of Data Items - Criteria

1. Every data item must have a system-wide unique name.
2. It should be possible to find the location of data items efficiently.
3. It should be possible to change the location of data items transparently.
4. Each site should be able to create new data items autonomously.
**Centralized Scheme - Name Server**

- **Structure:**
  - name server assigns all names
  - each site maintains a record of local data items
  - sites ask name server to locate non-local data items

- **Advantages:**
  - satisfies naming criteria 1-3

- **Disadvantages:**
  - does not satisfy naming criterion 4
  - name server is a potential performance bottleneck
  - name server is a single point of failure

**Use of Aliases**

- Alternative to centralized scheme: each site prefixes its own site identifier to any name that it generates i.e., site 17.account.
  - Fulfills having a unique identifier, and avoids problems associated with central control.
  - However, fails to achieve network transparency.

- Solution: Create a set of *aliases* for data items; Store the mapping of aliases to the real names at each site.
  - The user can be unaware of the physical location of a data item, and is unaffected if the data item is moved from one site to another.
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
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.

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 (2PC) protocol is widely used
- The three-phase commit (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.
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 \( S_p \) and let the transaction coordinator at \( S_j \) be \( C_i \).

**Phase 1: Obtaining a Decision**

- Coordinator asks all participants to **prepare** to commit transaction \( T \).
  - \( C_i \) adds the records `<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 `<no T>` to the log and send **abort** \( T \) message to \( C_i \).
  - if the transaction can be committed, then:
    - add the record `<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 if $C_i$ received a **ready** $T$ message from all the participating sites; otherwise $T$ must be aborted.
- Coordinator adds a decision record, `<commit $T$>` or `<abort $T$>`, to the log and forces record onto stable storage. Once the record is on 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 $S_k$ recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contains `<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.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
  - The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results.
Recovery and Concurrency Control

- **In-doubt transactions** have a `<ready T>`, but neither a `<commit T>`, nor an `<abort T>` log record.

- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.

- Recovery algorithms can note lock information in the log.
  - Instead of `<ready T>`, write out `<ready T, L> L = list of locks held by T when the log is written (read locks can be omitted).`
  - For every in-doubt transaction T, all the locks noted in the `<ready T, L>` log record are reacquired.

- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Alternative Models of Transaction Processing

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g. transaction crossing an organizational boundary
  - No organization would like to permit an externally initiated transaction to block local transactions for an indeterminate period

- Alternative models carry out transactions by sending messages
  - Code to handle messages must be carefully designed to ensure atomicity and durability properties for updates
    - Isolation cannot be guaranteed, in that intermediate stages are visible, but code must ensure no inconsistent states result due to concurrency
  - **Persistent messaging systems** are systems that provide transactional properties to messages
    - Messages are guaranteed to be delivered exactly once
    - Will discuss implementation techniques later
**Alternative Models (Cont.)**

- Motivating example: funds transfer between two banks
  - Two phase commit would have the potential to block updates on the accounts involved in funds transfer
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)

- Atomicity issue
  - Once transaction sending a message is committed, message must guaranteed to be delivered
    - Guarantee as long as destination site is up and reachable, code to handle undeliverable messages must also be available
      - E.g., credit money back to source account.
  - If sending transaction aborts, message must not be sent

**Error Conditions with Persistent Messaging**

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - E.g., if destination account does not exist, failure message must be sent back to source site
  - When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed
      - Get humans to take care of problem

- User code executing transaction processing using 2PC does not have to deal with such failures

- There are many situations where extra effort of error handling is worth the benefit of absence of blocking
  - E.g., pretty much all transactions across organizations
**Persistent Messaging and Workflows**

- **Workflows** provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps
  - E.g. when a bank receives a loan application, it may need to
    - Contact external credit-checking agencies
    - Get approvals of one or more managers
    and then respond to the loan application
  - Persistent messaging forms the underlying infrastructure for workflows in a distributed environment