What Is a Distributed Database?

- One in which data is:
  - partitioned or fragmented among several sites and/or
  - replicated – copies of the same data are made available at multiple sites

- It is managed by a distributed DBMS (or DDBMS) – DBMSs at two or more sites that jointly provide access to a single logical database.
What Is a Distributed Database? (cont.)

- Example:

![Diagram showing a network of sites with databases]

- A given site may have a local copy of all, part, or none of a particular database.
  - makes requests of other sites as needed

Fragmentation

- Divides up the records in a database among several sites.
  - the resulting "pieces" of the database are known as fragments

- Let R be a collection of records of the same type (e.g., a relation).

  - Horizontal fragmentation divides up the "rows" of R.
    - R(a, b, c) \rightarrow R1(a, b, c), R2(a, b, c), ...
    - R = R1 \cup R2 \cup ...

  - Vertical fragmentation divides up the "columns" of R.
    - R(a, b, c) \rightarrow R1(a, b), R2(a, c), ...
      (a is the primary key)
    - R = R1 \bowtie R2 \bowtie ...

Fragmentation (cont.)

- Another version of vertical fragmentation: divide up the tables (or other collections of records).
  - e.g., site 1 gets tables A and B
  - site 2 gets tables C and D

Example of Fragmentation

- Here's a relation from a centralized bank database:

<table>
<thead>
<tr>
<th>account</th>
<th>owner</th>
<th>street</th>
<th>city</th>
<th>branch</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111</td>
<td>E. Scrooge</td>
<td>1 Rich St</td>
<td></td>
<td>main</td>
<td>111111</td>
</tr>
<tr>
<td>123456</td>
<td>R. Cratchit</td>
<td>5 Poor Ln</td>
<td></td>
<td>west</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Here's one way of fragmenting it:
Replication

- Replication involves putting copies of the same collection of records at different sites.

<table>
<thead>
<tr>
<th>account type</th>
<th>interest rate</th>
<th>monthly fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>0%</td>
<td>$10</td>
</tr>
<tr>
<td>big saver</td>
<td>2%</td>
<td>$50</td>
</tr>
</tbody>
</table>

Reasons for Using a DDBMS

- to improve performance
  - how does distribution do this?

- to provide high availability
  - replication allows a database to remain available in the event of a failure at one site

- to allow for modular growth
  - add sites as demand increases
  - adapt to changes in organizational structure

- to integrate data from two or more existing systems
  - without needing to combine them
  - allows for the continued use of legacy systems
  - gives users a unified view of data maintained by different organizations
Challenges of Using a DDBMS (partial list)

- determining the best way to distribute the data
  - when should we use vertical/horizontal fragmentation?
  - what should be replicated, and how many copies do we need?

- determining the best way to execute a query
  - need to factor in communication costs

- maintaining integrity constraints (primary key, foreign key, etc.)

- ensuring that copies of replicated data remain consistent

- managing distributed txns: ones that involve data at multiple sites
  - need to still provide the ACID properties
  - atomicity and isolation can be harder to guarantee

Important Goal: Transparency

- Users shouldn't need to know:
  - that a database has been fragmented
  - where the fragments are located

- Instead, they are able to treat the database as a single logical database that is fully available at all of the sites.

- In addition, users shouldn't need to know about replication.
  - send their requests and updates to one site
  - they will be propagated to the replicas as needed

- It's possible to have a limited form of transparency.
  - example:
    - require fragment-specific names in queries
    - don't require knowledge of fragment locations or of the degree of replication
Failures in a DDBMS

- In addition to the failures that can occur in a centralized system, there are additional types of failures for a DDBMS.

- These include:
  - the loss or corruption of messages
    - TCP/IP handles this type of error
  - the failure of a site
  - the failure of a communication link
    - can often be dealt with by rerouting the messages
  - *network partition*: occurs if communication failures cause there to be no communication between two sets of sites in the system

Distributed Transactions

- A *distributed transaction* involves data stored at multiple sites.

- One of the sites serves as the *coordinator* of the transaction.
  - one option: the site on which the txn originated

- The coordinator divides a distributed transaction into *subtransactions*, each of which executes on one of the sites.

  - Each site maintains its own logs, recovery system, lock tables, …

  - The sites need to coordinate with each other to ensure that either all subtxns commit or all of them abort.
**Distributed Atomicity**

- To maintain atomicity in a distributed database:
  - before any site commits its subtxn, it must ensure all other sites are able to commit their subtxns.
  - if any site is unable to complete its subtxn, all of the sites must abort their subtxns.

- Example of what could go wrong:
  - a subtxn at one of the sites deadlocks and is aborted.
  - before the coordinator finds out about this, it notifies the other sites that it's time to commit, and they commit.

- Another example:
  - the coordinator notifies the other sites that it's time to commit.
  - most of the sites commit their subtxns.
  - one of the sites crashes before committing.

---

**Two-Phase Commit (2PC)**

- Two-phase commit is a protocol for deciding whether to commit or abort a distributed transaction.

- Basic idea:
  - coordinator asks sites if they're ready to commit.
  - if all sites say they're ready, all subtxns are committed.
  - otherwise, all subtxns are aborted.

- Before saying that it's ready, a site must ensure that its subtxn can be either committed or aborted – even after a failure.
  - what steps would be needed?

  - after these steps, the subtxn is in the *ready state*.
    - aka the *precommitted* state.
    - a site in the ready state must wait to be told what to do next.
2PC Phase I: Prepare

• When it's time to commit a distributedtxn T, the coordinator
  • force-writes a prepare record for T to its own log
  • sends a prepare message to each participating site

• If a site decides that it is ready to commit its subtxn, it:
  • takes the steps needed to put its txn in the ready state
  • force-writes a ready record for T to its log
  • sends a ready message for T to the coordinator and waits

• If a site decides that it needs to abort its subtxn, it:
  • force-writes a do-not-commit record for T to its log
  • sends a do-not-commit message for T to the coordinator
  • can it abort the subtxn now?

• Note: we always log a message before sending it to others.
  • allows the decision to send the message to survive a crash

2PC Phase II: Commit or Abort

• The coordinator reviews the messages from the sites.
  • if a site fails to send a message within some time interval,
    the coordinator assumes a do-not-commit message

• If all sites sent ready messages for T, the coordinator:
  • force-writes a commit record for T to its log
    • T is now officially committed
  • sends commit messages for T to the participating sites

• Otherwise, the coordinator:
  • force-writes an abort record for T to its log
  • sends abort messages for T to the participating sites

• Each site:
  • force-writes the appropriate record (commit or abort) to its log
  • commits or aborts its subtxn as instructed
2PC Phase II: Commit or Abort (cont.)

- Many implementations of 2PC include the following additions:
  - a site acknowledges its receipt of the commit/abort message by sending the coordinator an *ack message*
  - once the coordinator receives acks from all sites, it writes an end log record for the transaction

- The acks allow the coordinator to know when it can remove info. about a txn from its in-memory transaction table.

- These additions also allow the coordinator to know whether it should resend the commit/abort messages after recovering from a crash.

---

2PC State Transitions

- A subtxn can enter the aborted state from the initial state at any time.
- After entering the ready state, it can only enter the aborted state after receiving an abort message.
- A subtxn can only enter the committed state from the ready state, and only after receiving a commit message.
Recovery When Using 2PC

- The decision of whether to undo or redo a distributedtxn T is based on the last record for T in the log.

- If the last log record for T is anything but a ready record: treat T like any other txn.
  - if it's a commit record, redo T's updates as needed
  - otherwise, undo T's updates as needed
    - why does this work if the last record is do-not-commit?

  - what if the last record is from before 2PC began?

Recovery When Using 2PC (cont.)

- If the last log record for T is a ready record: contact the coordinator (or another site) to determine T's fate.
  - because the log records for T were forced to disk, the site can still commit or abort T as needed
  - if it can't reach another site, it must block until it can find out what to do!
What if the Coordinator Fails?

- The other sites can either:
  - wait for the coordinator to recover
  - elect a new coordinator

- If a participating site received a prepare message for a txn T from the failed coordinator but hadn't yet sent back a ready message, it can abort T now.
  - why is this always safe?

- this is preferable to trying to continue with the protocol, because it allows the fate of T to be decided

What if the Coordinator Fails? (cont.)

- If a participating site sent a ready message for T to the failed coordinator but didn't get back a commit or abort message, it can poll the other surviving sites about T's fate:
  - if at least one site has a commit record for T, T was committed
  - if at least one site has an abort record for T, T was aborted
  - if no site has either a commit or abort record and at least one site does not have a ready record for T, T can be aborted now
  - if no site has either a commit or abort record, and all surviving sites have ready records for T, can't determine T's fate unless the failed coordinator recovers!
    - why?
2PC Example

- T requires subtxns at sites A, B, C. Site A is the coordinator.
- After the last operation in T, site A:
  - sends prepare messages to B and C
  - puts its own subtxn in the ready state.
- Sites B and C:
  - put their subtxns in the ready state
  - send ready messages to A
- Site A:
  - writes its commit record (T is now officially committed)
  - sends commit messages to B and C
- B writes its commit record.
- C crashes before writing its commit record. What happens?

Types of Replication

- In synchronous replication, transactions are guaranteed to see the most up-to-date value of an item.
- Because synchronous replication is often too slow, asynchronous replication is often used in practice.
  - transactions are not guaranteed to see the latest value
Synchronous Replication I: Read-Any, Write-All

- **Read-Any**: when reading an item, access any of the replicas.
- **Write-All**: when writing an item, must update all of the replicas.
- Works well when reads are much more frequent than writes.
- Drawback: writes are very expensive.

Synchronous Replication II: Voting

- When writing, update some fraction of the replicas.
- When reading, read enough copies to ensure you get at least one copy of the most recent value.
  - use a version number to determine which value is most recent
  - the copies "vote" on the value of the item
- Drawback: reads are now more expensive
- How many copies must be read?
  - let: \( n \) = the number of copies
  \( w \) = the number of copies that are written
  \( r \) = the number of copies that are read
  - need: \( r > n - w \) (i.e., at least \( n - w + 1 \)) -- but see later notes!
- example: 6 replicas
  - update 4
  - must read at least 3 (not necessarily a majority!)
Distributed Concurrency Control

- To ensure the isolation of distributed transactions, need some form of distributed concurrency control.

- Extend the concurrency control schemes that we studied earlier.
  - We'll focus on extending strict 2PL

- If we just used strict 2PL at each site, we would ensure that the schedule of subtxns at each site is serializable.
  - Why isn't this sufficient?

Distributed Concurrency Control (cont.)

- Example of why special steps are needed:
  - Voting-based synchronous replication with 6 replicas
  - Let's say that we configure the voting as follows:
    - Each write updates 3 copies
    - Each read accesses 4 copies
  - Can end up with schedules that are not conflict serializable

- Example:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xl(A1); xl(A2); xl(A3)$</td>
<td>$xl(A4); xl(A5); xl(A6)$</td>
</tr>
<tr>
<td>$w(A1); w(A2); w(A3)$</td>
<td>$w(A4); w(A5); w(A6)$</td>
</tr>
<tr>
<td>$xl(B1); xl(B2); xl(B3)$</td>
<td>$xl(B4); xl(B5); xl(B6)$</td>
</tr>
<tr>
<td>$w(B1); w(B2); w(B3)$</td>
<td>$w(B4); w(B5); w(B6)$</td>
</tr>
</tbody>
</table>

$Xi =$ the copy of item X at site $i$

$T1$ should come before $T2$ based on the order in which they write A.

$T1$ should come after $T2$ based on the order in which they write B.
What Do We Need?

• We need a txn to be able to acquire a lock for a *logical item*, not just for individual copies of that item.
  • this doesn't necessarily mean locking every copy
  • it means preventing someone else from reading and/or writing what is considered the current value of the item according to whatever replication scheme is being used
  • we say that a transaction acquires a *global lock* for the item

• Requirements for global locks:
  • no two transactions can hold a global exclusive lock for the same item
  • any number of txns can hold a global shared lock for an item
  • a txn cannot acquire a global exclusive lock on an item if another txn holds a global shared lock on that item, and vice versa

What Do We Need? (cont.)

• In addition, we need to take appropriate steps to ensure the proper ordering of operations within a given distributed txn.
  • don't want a subtxn to get ahead of where it should be in the context of the txn as a whole
  • relevant even in the absence of replication
  • one option: have the coordinator of the txn acquire the necessary locks before sending operations to a site
Option 1: Centralized Locking

- In *centralized locking*, one site manages the lock requests for all items in the distributed database.
  - even items that do not have copies stored at that site
  - since there is only one place to acquire locks, these locks are obviously global locks

- Problems with this approach?

Option 2: Primary-Copy Locking

- *Primary-copy locking* is a variation on centralized locking.

- To prevent one site from becoming a bottleneck, we divide up the responsibility for managing lock requests among the sites in the DDBMS.

- One copy of an item is designated the *primary copy* of that item.

- The site on which the primary copy of an item resides is responsible for handling all lock requests on that item.
  - locking the primary copy of an item gives you a global lock on the item
Option 3: Fully Distributed Locking

• In *fully distributed locking*:
  • no one site is responsible for managing lock requests for a given item
  • a transaction acquires a global lock for an item by locking a sufficient number of the item’s copies
  • these local locks combine to form a global lock

Option 3: Fully Distributed Locking (cont.)

• How many copies must be locked?
  • let: \( n \) = the total number of copies
  • let: \( x \) = the number of copies that must be locked to acquire a global exclusive lock
  • let: \( s \) = the number of copies that must be locked to acquire a global shared lock
  • we need \( x > n/2 \)
  • guarantees that no two txns can both acquire a global exclusive lock at the same time
  • we need \( s > n - x \) (i.e., \( s + x > n \))
  • if there’s a global exclusive lock on an item, there aren’t enough unlocked copies for a global shared lock
  • if there’s a global shared lock on an item, there aren’t enough unlocked copies for a global excl. lock
Option 3: Fully Distributed Locking (cont.)

- Our earlier example would no longer be possible:

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xl(A₁); xl(A₂); xl(A₃)</td>
<td>xl(A₄); xl(A₅); xl(A₆)</td>
</tr>
<tr>
<td></td>
<td>w(A₁); w(A₂); w(A₃)</td>
<td>w(A₄); w(A₅); w(A₆)</td>
</tr>
<tr>
<td></td>
<td>xl(B₁); xl(B₂); xl(B₃)</td>
<td>xl(B₄); xl(B₅); xl(B₆)</td>
</tr>
<tr>
<td></td>
<td>w(B₁); w(B₂); w(B₃)</td>
<td>w(B₄); w(B₅); w(B₆)</td>
</tr>
</tbody>
</table>

Synchronous Replication and Fully Distributed Locking

- Read-any write-all:
  - when writing an item, a txn must update all of the replicas
    - this gives it \( x = n \) exclusive locks, so \( x > n/2 \)
  - when reading an item, a txn can access any of the replicas
    - this gives it \( s = 1 \) shared lock, and \( 1 > n - n \)

- Voting:
  - when writing, a txn updates a *majority of the copies* – i.e., \( w \) copies, where \( w > n/2 \).
    - this gives it \( x > n/2 \) exclusive locks as required
  - when reading, a txn reads \( r > n - w \) copies
    - this gives it \( s = n - x \) shared locks as required
Distributed Deadlock Handling

• Under centralized locking, we can just use one of the schemes that we studied earlier in the semester.

• Under the other two locking schemes, deadlock detection becomes more difficult.
  • local waits-for graphs alone will not necessarily detect a deadlock
    • example:
      site 1: [diagram]
      site 2: [diagram]

      • one option: periodically send local waits-for graphs to one site that is responsible for performing global deadlock detection

• Instead of using deadlock detection, it is often easier to use the timeout-based scheme that we covered earlier.
  • if a txn waits for too long, presume deadlock and abort it

Asynchronous Replication I: Primary Site

• In primary-site replication, one replica is designated the primary or master replica.

• Only the master is updatable.

• The other replicas (the secondaries or clients) can only be read.
  • no locks are acquired when accessing the clients
  • thus, we only use them when performing read-only txns

• Changes to the master are periodically propagated to the clients.
  • need to ensure they are applied in such a way that you always have a consistent snapshot

• Drawbacks of this approach?
Asynchronous Replication II: Peer-to-Peer

- In *peer-to-peer replication*, more than one replica is updatable.
- Problem: need to somehow resolve conflicting updates!