Distributed Databases and Replication

Computer Science E-66
Harvard University
David G. Sullivan, Ph.D.

What Is a Distributed Database?

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

- It is managed by a distributed DBMS (DDBMS) – processes on two or more machines that jointly provide access to a single logical database.

- The machines in question may be:
  - at different locations (e.g., different branches of a bank)
  - at the same location (e.g., a cluster of machines)

- In the remaining slides, we will use the term site to mean one of the machines involved in a DDBMS.
  - may or may not be at the same location
A given site may have a local copy of all, part, or none of a particular database. It makes requests of other sites as needed.

**Fragmentation / Sharding**

- Divides up a database’s records among several sites
  - the resulting “pieces” are known as *fragments/shards*
- Let R be a collection of records of the same type (e.g., a relation).
  - **Horizontal fragmentation** divides up the "rows" of R.
    - $R(a, b, c) \rightarrow R_1(a, b, c), R_2(a, b, c), \ldots$
    - $R = R_1 \cup R_2 \cup \ldots$
  - **Vertical fragmentation** divides up the "columns" of R.
    - $R(a, b, c) \rightarrow R_1(a, b), R_2(a, c), \ldots$ (a is the primary key)
    - $R = R_1 \bowtie R_2 \bowtie \ldots$
Fragmentation / Sharding (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>$11111</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>
<tr>
<td>222222</td>
<td>R. Cratchit</td>
<td>5 Poor Ln</td>
<td>...</td>
<td>west</td>
<td>$70000</td>
</tr>
<tr>
<td>333333</td>
<td>R. Cratchit</td>
<td>5 Poor Ln</td>
<td>...</td>
<td>west</td>
<td>$70000</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>bigsaver</td>
<td>2%</td>
<td>$50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</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
  - atomicity and isolation can be harder to guarantee

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: failures prevent communication between two subgroups of the sites
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.

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

```
subtxn 1
```

```
read balance1
write(balance1 - 500)
```

```
subtxn 2
```

```
read balance2
write(balance2 + 500)
```

Types of Replication

- In synchronous replication, transactions are guaranteed to see the most up-to-date value of an item.
- In asynchronous replication, transactions are not guaranteed to see the most up-to-date 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.
  - each value has a version number that is increased when the value is updated
- When reading, read enough copies to ensure you get at least one copy of the most recent value (see next slide).
  - the copies "vote" on the value of the item
  - the copy with the highest version number is the most recent
- Drawback: reads are now more expensive
Synchronous Replication II: Voting (cont.)

- 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 \( n - w + 1 \))
  - example:  
    - \( n = 6 \) copies
    - update \( w = 3 \) copies
    - must read at least 4 copies

- Example: 6 copies of data item A, each with value = 4, version = 1.
  - txn 2 updates A1, A2, and A4 to be 6 (and their version number becomes 2)
  - txn 1 reads A2, A3, A5, and A6
  - A2 has the highest version number (2), so its value (6) is the most recent.

Which of these allow us to ensure that clients always get the most up-to-date value?

- 10 replicas – i.e., 10 copies of each item

- voting-based approach with the following requirements:

<table>
<thead>
<tr>
<th>number of copies accessed when reading</th>
<th>number of copies accessed when writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. 7</td>
<td>3</td>
</tr>
<tr>
<td>B. 5</td>
<td>5</td>
</tr>
<tr>
<td>C. 9</td>
<td>2</td>
</tr>
<tr>
<td>D. 4</td>
<td>8</td>
</tr>
</tbody>
</table>

(select all that work)
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 before:
    • 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); w(A1); w(A2); w(A3)</td>
<td>xl(A4); xl(A5); xl(A6); w(A4); w(A5); w(A6)</td>
</tr>
<tr>
<td>xl(B1); xl(B2); xl(B3); w(B1); w(B2); w(B3)</td>
<td>xl(B4); xl(B5); xl(B6); w(B4); w(B5); w(B6)</td>
</tr>
</tbody>
</table>

Xi = the copy of item X at site i

T_1 should come before T_2 based on the order in which they write A.

T_1 should come after T_2 based on the order in which they write B.
What Do We Need?

- We need shared and exclusive locks for a *logical item*, not just for individual copies of that item.
  - referred to as *global locks*
  - doesn't necessarily mean locking every copy

- Requirements for global locks:
  - no two txns 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 ensure the correct ordering of operations within each distributed transaction.
  - 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

- One site manages the lock requests for all items in the distributed database.
  - even items that don't have copies stored at that site
  - since there's only one place to acquire locks, these locks are obviously global locks!

- Problems with this approach:
  - the lock site can become a bottleneck
  - if the lock site crashes, operations at all sites are blocked

Option 2: Primary-Copy Locking

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

- The site holding the primary copy handles all lock requests for that item.
  - acquiring a shared lock for the primary copy gives you a global shared lock for the item
  - acquiring an exclusive lock for the primary copy gives you a global exclusive lock for the item

- To prevent one site from becoming a bottleneck, distribute the primary copies among the sites.

- Problem: If a site goes down, operations are blocked on all items for which it holds the primary copy.
Option 3: 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 the global lock

• To acquire a global shared lock, acquire local shared locks for a sufficient number of copies (see next slide).

• To acquire a global exclusive lock, acquire local exclusive locks for a sufficient number of copies (see next slide).

Option 3: Fully Distributed Locking (cont.)

• How many copies must be locked?
  • let:  \( n \) = the total number of copies
    \( x \) = the number of copies that must be locked to acquire a global exclusive lock
    \( s \) = 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>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A₁);</td>
<td>xl(A₄); xl(A₅); xl(A₆);</td>
</tr>
<tr>
<td>xl(A₂);</td>
<td>xl(A₄); xl(A₅); xl(A₆);</td>
</tr>
<tr>
<td>xl(A₃);</td>
<td>xl(A₄); xl(A₅); xl(A₆);</td>
</tr>
<tr>
<td>xl(B₁);</td>
<td>xl(B₄); xl(B₅); xl(B₆);</td>
</tr>
<tr>
<td>xl(B₂);</td>
<td>xl(B₄); xl(B₅); xl(B₆);</td>
</tr>
<tr>
<td>xl(B₃);</td>
<td>xl(B₄); xl(B₅); xl(B₆);</td>
</tr>
<tr>
<td>w(B₁);</td>
<td>w(B₄); w(B₅); w(B₆)</td>
</tr>
<tr>
<td>w(B₂);</td>
<td>w(B₄); w(B₅); w(B₆)</td>
</tr>
<tr>
<td>w(B₃)</td>
<td>w(B₄); w(B₅); w(B₆)</td>
</tr>
</tbody>
</table>

- n = 6
- need x > 6/2
- must acquire at least 4 local exclusive locks before writing

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
Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- *fully distributed* locking
- voting-based approach with the following requirements:

<table>
<thead>
<tr>
<th>number of copies</th>
<th>read</th>
<th>written</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>B.</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>C.</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>D.</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(select all that work)

Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- *primary-copy* locking
- voting-based approach with the following requirements:

<table>
<thead>
<tr>
<th>number of copies</th>
<th>read</th>
<th>written</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>B.</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>C.</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>D.</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(select all that work)
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:
      
      ![Diagram showing two sites with transactions T1 and T2]

    - one option: periodically send local waits-for graphs to one site that checks for deadlocks

- Instead of using deadlock detection, it's often easier to use a timeout-based scheme.
  - if atxn waits too long, presume deadlock and roll it back!

---

Recall: Types of Replication

- In *synchronous replication*, transactions are guaranteed to see the most up-to-date value of an item.

- In *asynchronous replication*, transactions are *not* guaranteed to see the most up-to-date value.
Asynchronous Replication I: Primary Site

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

• All writes go to the primary.
  • propagated asynchronously to the other replicas (the secondaries)

• The secondaries can only be read.
  • no locks are acquired when accessing them
  • thus, we only use them when performing read-only txns

• Drawbacks of this approach?

Asynchronous Replication II: Peer-to-Peer

• In peer-to-peer replication, more than one replica can be updated.

• Problem: need to somehow resolve conflicting updates!
Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- fully distributed locking
- voting-based approach with the following requirements:

<table>
<thead>
<tr>
<th>number of copies</th>
<th>read</th>
<th>written</th>
<th>requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>5</td>
<td>5</td>
<td>yes</td>
</tr>
<tr>
<td>B.</td>
<td>6</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>C.</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D.</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

(select all that work)
Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- fully distributed locking
- voting-based approach with the following requirements:

<table>
<thead>
<tr>
<th>number of copies</th>
<th>read</th>
<th>written</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>B.</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>C.</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>D.</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(select all that work)

### Voting-based Approach

| A. 5 5 | yes |
| B. 6 4 | no  |
| C. 7 3 | no  |
| D. 4 5 | no  |

**Problem:** two txns can both get a global exclusive lock at the same time!