Distributed Databases

Chapter 22.6-22.14
Final Exam

- When Monday, May 3, at 4pm
- Where, here FB007
- What
  - Open book, open notes, no computer
  - 48-50 multiple choice questions
  - 180 minutes
  - Approx. 1/3 covering material since last quiz (Lectures 22-26), remaining 2/3 Lectures 1-21
Distributed Databases

- Data is stored at several sites, each managed by a DBMS that runs independently.
- **Distributed Data Independence:** Users should not have to know where data is located (extends Physical and Logical Data Independence principles).
- **Distributed Transaction Atomicity:** Users should be able to write Xacts accessing multiple sites just like local Xacts.
- Hard to achieve; weakened in practice
Types of Distributed Databases

- **Homogeneous**: Every site runs the same type of DBMS.
- **Heterogeneous**: Different sites run different DBMSs (different RDBMSs or even non-relational DBMSs).

Gateway protocols
Distributed DBMS Architectures

- **Client-Server**
  - Client ships query to single site. All query processing at servers.
  - Set-oriented communication, client side caching.
  - Client is aware of where data lives

- **Collaborating-Servers**
  - Query spans multiple sites.
Storing Data

- **Fragmentation**
  - **Horizontal decomposition:** Often disjoint.
  - **Vertical decomposition:** Lossless-join; tids.

- **Replication**
  - Increases data availability.
  - Faster query evaluation.
  - **Synchronous vs. Asynchronous.**
    - Vary in how copies are kept current.
  - Can be combined with caching
Distributed Catalog Management

- Must keep track or estimate of how data is distributed across sites.
- Must be able to name each replica of each fragment. To preserve local autonomy:
  - `<local-name, birth-site>`

Site Catalog: Describes all objects (fragments, replicas) at a site + Keeps track of copies of relations created at this site.
- To find a relation, look up its birth-site catalog.
- Birth-site never changes, even if relation is moved.
Distributed Queries

- Horizontally Fragmented:
  Suppose tuples with rating < 5 at Shanghai, and >= 5 at Tokyo.
  - Could compute \( \text{SUM}(\text{age}) \), \( \text{COUNT}(\text{age}) \) at both sites and then combine.
  - If WHERE contained just \( \text{S.rating} > 6 \), just one site.

- Even if tuples were fragmented by \( \text{age} \) or \( \text{location} \) this would be a relatively easy query to distribute

```sql
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
AND S.rating < 7
```
Distributed Queries

- Vertically Fragmented: sid and rating at Shanghai, sid, sname and age at Tokyo,
  - Must reconstruct relation by join on sid, then evaluate the query.
  - Pushing a select on rating and projection on sid to Shanghai, would help this query

- Replicated: Sailors copies at both sites.
  - Choice of site based on local costs, shipping costs.
Distributed Joins

- Fetch as Needed, Page-Nested Loop, Sailors as outer loop:
  - **Cost:** $500 \text{D} + 500 \times 1000 \text{(D+S)}$
  - \text{D} is cost to read/write page; \text{S} is cost to ship page.
  - If query was not submitted at London, must add cost of shipping result to query site.
  - Can also do INL at London, fetching matching Reserves tuples to London as needed.
  - Server caching helps, $(500 + 500\times1000)\text{(D)} + 1000\text{(S)}$ if there is enough space for Reserves
Distributed Joins

- **Ship to One Site:**
  - Ship Reserves to London.
    - Cost: 1000 S + 4500 D
      - (Assumes Sort-Merge Join with adequate buffer pages; cost = 3 Passes*(500+1000))
    - Pushing selects and projections to remote server can offer large benefits
    - External Sorts can be distributed (parallelism)
    - If result size is very large, may be better to ship both relations to result site and then join them!
Semijoin

- At London, Push selection to Sailors, project Sailors onto common join columns of reserves (sid) and ship this to Paris.
- At Paris, Apply join of Sailors projection with Reserves.
  - Result is called a reduction of Reserves wrt Sailors.
- Ship reduction of Reserves back to London.
- At London, join Sailors with reduction of Reserves.
- Idea: Tradeoff the cost of computing and shipping projection for cost of shipping full Reserves relation.
- Especially useful if there is significant filtering due to the selection on Sailors.
Bloomjoin

- **At London**, compute a bit-vector of some size $k$:
  - Hash join-column values ($sid$) into range 0 to $k-1$.
  - If any tuple hashes to $I$, set bit $I$ to 1 ($I$ from 0 to $k-1$).
  - Ship this bit-vector to Paris.

- **At Paris**, hash each tuple of Reserves similarly, and discard tuples that hash to 0 in Sailors bit-vector.
  - Result is called **reduction** of Reserves wrt Sailors.

- Ship bit-vector reduced Reserves to London.
- **At London**, join Sailors with reduced Reserves.
- Bit-vector cheaper to ship, almost as effective.
Distributed Query Optimization

- Cost-based approach; consider a large set of plans, pick cheapest; similar to centralized optimization.
  - **Difference 1:** Consider communication costs.
  - **Difference 2:** Remote site autonomy must be respected (how much load can we push elsewhere).
  - **Difference 3:** Incorporate new distributed join methods.

- Query site constructs **global plan**, with **suggested local plans** describing processing at each site.
  - If a site can improve suggested local plan, free to do so.
Updating Distributed Data

- **Synchronous Replication**: All copies of a modified relation (fragment) must be updated before the modifying Xact commits.
  - Data distribution is made transparent to users.
- **Asynchronous Replication**: Copies of a modified relation are only periodically updated; different copies may get out of synch in the meantime.
  - Users must be aware of data distribution.
  - Current products follow this approach.
Synchronous Replication

- **Voting (Read-most, Write-most):** Xact must write a majority of copies to modify an object; must read enough copies to ensure it sees the most recent copy.
  - E.g., 10 copies; 7 written for update; 6-10 copies read.
  - Each copy has version number.
  - Not attractive usually because reads are common.

- **Read-any, Write-all:** Writes are slower and reads are faster, relative to Voting.
  - Most common approach to synchronous replication.

- **Choice of technique determines which locks to set.**
Cost of Synchronous Replication

- Before an modifying Xact can commit, it must obtain locks on all copies. Sends lock requests to remote sites, and while waiting for the response, it holds on to other locks!

- Alternatively, it can lock one copy, delete all others (which must be unlocked), update it, and release lock.

- However, if sites or links fail, Xact cannot commit until they are back up.

- Even if there is no failure, committing must follow an expensive commit protocol with many msgs.

- So the alternative of asynchronous replication is becoming widely used.
Asynchronous Replication

- Allows modifying Xact to commit before all copies have been changed or deleted (and readers nonetheless look at just one copy).
- Users must be aware that copies may be out-of-sync for short periods of time.
- Two approaches: **Primary Site** and **Peer-to-Peer** replication.
  - Difference lies in how many copies are “updatable” or “master copies”.
Peer-to-Peer Replication

- More than one (or all) of the copies of an object can be “updateable” (the “master-copy”).
- Changes to a “master-copy” are propagated to other copies.
- If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe’s age changed to 35; Site 2: to 36)
- Best used when conflicts do not arise often:
  - E.g., Each master site owns a disjoint fragment.
  - E.g., Updating rights owned by one master at a time.
Primary Site Replication

- Exactly one copy of a relation is designated the primary or master-copy. Replicas at other sites cannot be directly updated.
  - The primary copy is published.
  - Other sites subscribe to (fragments of) this relation; these are secondary copies.
- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps. First, capture changes made by committed Xacts; then apply these changes.
Implementing the Capture Step

- **Log-Based Capture:** The log (kept for recovery) is used to generate a Change Data Table (CDT).
  - If this is done when the log tail is written to disk, must somehow remove changes due to subsequently aborted Xacts.

- **Procedural Capture:** A procedure that is automatically invoked (trigger) does the capture; typically, just takes a snapshot (copies the command onto an Change list).

- Log-Based Capture is better (cheaper, faster) but relies on proprietary log details.
Implementing the Apply Step

- The Apply process at the secondary site periodically obtains (a snapshot or) changes to the CDT table from the primary site, and updates the copy.
  - Period can be timer-based or user/application defined.
  - Can be either pushed by primary, or pulled by secondary

- Log-Based Capture plus continuous Apply minimizes delay in propagating changes.

- Procedural Capture plus application-driven Apply is the most flexible way to process changes.
Data Warehousing and Replication

- **A hot trend**: Building giant “warehouses” of data consolidated from many sites.
  - Enables complex decision support queries over data from across an organization.

- Warehouses are an instance of asynchronous replication with infrequent updates.
  - Source data typically controlled by different DBMSs; emphasis on “cleaning” data and removing mismatches ($ vs. rupees) while creating replicas.

- Procedural capture and application Apply best for this environment.
Distributed Locking

How do we manage locks for objects across many sites?

- **Centralized:** One site does all locking.
  - Vulnerable to single site failure.

- **Primary Copy:** All locking for an object done at the primary copy site for this object.
  - Reading requires access to locking site as well as site where the object is stored.

- **Fully Distributed:** Locking for a copy done at site where the copy is stored.
  - Locks at all sites while writing an object.
Distributed Deadlock Detection

- Each site maintains a local waits-for graph.
- A global deadlock might exist even if the local graphs contain no cycles:

  SITE A:
  - T1 → T2

  SITE B:
  - T1 ← T2

  GLOBAL:
  - T1 ← T2

- Three solutions: Centralized (send all local graphs to one site); Hierarchical (organize sites into a hierarchy and send local graphs to parent in the hierarchy); Timeout (abort Xact if it waits too long).
Summary

- Distributed DBMSs offer site autonomy and distributed administration.
- Must design how data is distributed
  - Horizontally Fragmented
  - Vertically Fragmented
  - Replicated
- Distributed plans need to be considered in Query optimization
- Must revisit storage and catalog techniques, concurrency control, and recovery issues.