**Topics**
- Distributed DBMS architectures
- Data storage in a distributed DBMS
- Distributed catalog management
- Distributed query processing
- Updates in a distributed DBMS
- Distributed transaction management

**Introduction**
- Data is stored at several sites, each managed by a DBMS that can run independently.
- Data distribution is transparent.

**Desirable Properties**
- 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.

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

Middleware systems: Extra layer between client and server.
- Middleware usually does not store data, but is capable of performing some relational operators

**Collaborating Servers:** Queries can span multiple sites.

**Data Storage**

Two main new techniques for storing a relation:
- Fragmentation: Partition the relation into small parts called fragments
  - Goal: Locality
  - Horizontal versus vertical fragmentation
- Replication: Store several copies of a relation or relation fragment
  - Goals: Increased availability, locality

**Storing Data**

- Fragmentation
  - Horizontal: Usually disjoint.
  - Vertical: Lossless-join; tids.
- Replication
  - Gives increased availability.
  - Faster query evaluation.
  - Synchronous vs. asynchronous.
  - Vary in how current copies are.

**Distributed Catalog Management**

- Must keep track 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 and keeps track of replicas 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 Query Processing**

What is different?
- Queries can involve several sites
- New cost model (communication cost might be significant)
- New operator implementations for queries involving several sites
**Distributed Queries**

- Horizontally Fragmented:
  - Tuples with rating < 5 at Shanghai, >= 5 at Tokyo.
  - Must compute `SUM(age)`, `COUNT(age)` at both sites.
  - If `WHERE` contained just `S.rating>=6`, just one site.
- Vertically Fragmented:
  - `<>(tid, sid, rating)` at Shanghai, `<>(tid, sname, age)` at Tokyo.
  - Must reconstruct relation by join on `tid`, then evaluate the query.
- Replicated: Sailors copies at both sites.
  - Choice of site based on local costs, shipping costs.

**Distributed Joins**

- Fetch as Needed, Page NL, Sailors as outer:
  - Cost: `500 D + 500 * 1000 (D+S)`
  - `D` is cost to read/write page; `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.

- Ship to One Site: Ship Reserves to London.
  - Cost: `1000 S + 4500 D` (SM Join; cost = `3*(500+1000)`)  
  - If result size is very large, may be better to ship both relations to result site and then join them!

- Idea: Tradeoffs
  1. Cost of computing and shipping the projection versus
  2. Cost of shipping full Reserves relation.

- Especially useful if there is a selection on Sailors, and answer desired at London

**Distributed Joins: Semijoin**

- At London, project Sailors onto join columns and ship this to Paris.
- At Paris, join Sailors projection with Reserves.
  - Result is called reduction of Reserves wrt Sailors.
  - Ship reduction of Reserves to London.
- At London, join Sailors with reduction of Reserves.

**Distributed Joins: Bloomjoin**

- At London, compute a bit-vector of some size `k` (usually `k < #records`):
  - Hash join column values into range `0` to `k-1`.
  - If some record hashes to bit 1, set bit 1 to 1 (if from `0` to `k-1`).
  - Ship bit-vector to Paris.
- At Paris, hash each record of Reserves similarly, and discard records that hash to 0 in Sailors bit-vector.
  - Result is called reduction of Reserves wrt. Sailors.

**Distributed Joins: Bloomjoin**

- Ship bit-vector “reduced Reserves” from Paris to London.
- At London, join Sailors with reduced Reserves.
- Bit-vector cheaper to ship, almost as effective.
**Distributed Query Optimization**
- Cost-based approach; consider all plans, pick cheapest; similar to centralized optimization.
  - Difference 1: Communication costs must be considered.
  - Difference 2: Local site autonomy must be respected.
  - Difference 3: 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.

**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
  - SITE B
  - GLOBAL
- 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).

**Distributed Recovery**
- Two new issues:
  - New kinds of failure, e.g., links and remote sites.
  - If “sub-transactions” of an Xact execute at different sites, all or none must commit. Need a commit protocol to achieve this.
- A log is maintained at each site, as in a centralized DBMS, and commit protocol actions are additionally logged.

**Two-Phase Commit (2PC)**
- Site at which Xact originates is coordinator; other sites at which it executes are subordinates.
- When an Xact wants to commit:
  1. Coordinator sends prepare msg to each subordinate.
  2. Subordinate force-writes an abort or prepare log record and then sends a no or yes msg to coordinator.
  3. If coordinator gets unanimous yes votes, force-writes a commit log record and sends commit msg to all subs. Else, force-writes abort log rec, and sends abort msg.
  4. Subordinates force-write abort/commit log rec based on msg they get, then send ack msg to coordinator.
  5. Coordinator waits end log rec after getting all acks.

**Two-Phase Commit**
- Coordinator
  - Send prepare
    - Force-write prepare record
    - Send yes or no
  - Wait for all responses
    - Force-write commit or abort
    - Send commit or abort
    - Force-write abort or commit
    - Send ACK
  - Wait for all ACKs
    - Write end record
**Comments on 2PC**

❖ Two rounds of communication: First, voting; then, termination. Both initiated by coordinator.
❖ Any site can decide to abort an Xact.
❖ Every msg reflects a decision by the sender; to ensure that this decision survives failures, it is first recorded in the local log.
❖ All commit protocol log recs for an Xact contain Xactid and Coordinatorid. The coordinator’s abort/commit record also includes ids of all subordinates.

**Restart After a Failure at a Site**

❖ If we have a commit or abort log rec for Xact T, but not an end rec, must redo/undo T.
   – If this site is the coordinator for T, keep sending commit/abort msgs to subs until acks received.
❖ If we have a prepare log rec for Xact T, but not commit/abort, this site is a subordinate for T.
   – Repeatedly contact the coordinator to find status of T, then write commit/abort log rec; redo/undo T, and write end log rec.
❖ If we don’t have even a prepare log rec for T, unilaterally abort and undo T.
   – This site may be coordinator! If so, subs may send msgs.

**Blocking**

❖ If coordinator for Xact T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers.
   – T is blocked.
   – Even if all subordinates know each other (extra overhead in prepare msg) they are blocked unless one of them voted no.

**Link and Remote Site Failures**

❖ If a remote site does not respond during the commit protocol for Xact T, either because the site failed or the link failed:
   – If the current site is the coordinator for T, should abort T.
   – If the current site is a subordinate, and has not yet voted yes, it should abort T.
   – If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds.

**Observations on 2PC**

❖ Ack msgs used to let coordinator know when it can “forget” an Xact; until it receives all acks, it must keep T in the Xact Table.
❖ If coordinator fails after sending prepare msgs but before writing commit/abort log recs, when it comes back up it aborts the Xact.
❖ If a subtransaction does no updates, its commit or abort status is irrelevant.

**2PC with Presumed Abort**

❖ When coordinator aborts T, it undoes T and removes it from the Xact Table immediately.
   – Doesn’t wait for acks; “presumes abort” if Xact not in Xact Table. Names of subs not recorded in abort log rec.
❖ Subordinates do not send acks on abort.
❖ If subxact does not do updates, it responds to prepare msg with reader instead of yes/no.
❖ Coordinator subsequently ignores readers.
❖ If all subxacts are readers, 2nd phase not needed.