1 Distributed Databases

Distributed Databases

- Goal: a small-scale (dozens of sites) distributed DBMS
- Desiderata (or Extending data independence)
  - Location transparency
    - users don’t need know where data is
  - Performance transparency
    - performance should be independent of query submission site
  - Copy transparency
    - allow replication for availability
  - Transaction transparency
    - looks like single-site xacts
  - Fragment transparency
    - tables can be fragmented to different sites
    - like parallel databases
  - Schema change transparency
    - schema updates at a single site should affect global schema
  - Local DBMS transparency
    - arbitrary local DBMSs

- Many research prototypes: R*, SDD-1, Distributed Ingres etc...
- Few commercial systems...

*Partially based on notes from Joe Hellerstein
Distributed Databases

- ... vs Data Integration
  - Distributed data sources but query processing centralized
  - Query processing/performance easy to do
  - Harder questions: Schema integration/mapping

- ... vs Federated
  - Federated: A loose federation of autonomous sites
  - Emphasis on “autonomous”
  - Little control over what happens where
    - Mariposa Project @ Berkeley

- ... vs Grid/P2P
  - Complex query processing hard, not well motivated
  - PIER Project @ Berkeley: Allowed relations to be fragmented across DHT/Grid

R*

- Location transparency
- Local privacy & control
- Local performance on local queries
- Site autonomy: No central catalogs, no central scheduler, no central deadlock detector
- Catalogs stored as “soft state”
- Tables could be fragmented, replicated

2 Distributed Transactions

Distributed Transactions

- May make updates at many sites

- How do you commit ?
  - Everyone must commit or abort
  - 2-Phase, 3-Phase Commit Protocols

- Deadlocks (assuming locking is used) ?
  - Distributed deadlock detection
  - Solution 1: Pass around “waits-for” graphs
  - Solution 2: Time-out based (perhaps more practical)
Distributed Transactions in R*

- Assumptions
  - update in place, WAL
  - batched force of log records

- Desired Characteristics
  - guaranteed xact atomicity
  - ability to “forget” outcome of commit ASAP
  - minimal log writes & message traffic
  - optimized performance in no-failure case
  - exploitation of completely or partially R/O xacts
  - maximize ability to perform unilateral abort

- In order to minimize logging and comm:
  - rare failures do not deserve extra overhead in normal processing (“make common case fast”)
  - hierarchical commit better than 2P

Normal 2-Phase Commit

<table>
<thead>
<tr>
<th>Coordinator Log</th>
<th>Messages</th>
<th>Subordinate Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREPARE →</td>
<td></td>
<td>prepare*/abort*</td>
</tr>
<tr>
<td>← VOTE YES/NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit*/abort*</td>
<td>COMMIT/ABORT→</td>
<td>commit*/abort*</td>
</tr>
<tr>
<td></td>
<td>← ACK</td>
<td></td>
</tr>
</tbody>
</table>

- * → forced on log (for durability/atomicity)
- Always log before sending a message

- Total cost:
  - subords: 2 forced log-writes (prepare/commit), 2 messages (YES/ACK)
  - coord: 1 forced log write (commit), 1 async log write (end), 2 messages/subord (prepare/commit)

Normal 2-Phase Commit

- Dealing with failures
  - Recovery process at each site
  - Use logs for handling failures
– E.g. a subordinate crashes, and wakes up, and sees a prepare* → ask coordinator what happened to the transaction
– Need to prove correctness carefully

• Hierarchical 2PC
  – Tree-shaped communication hierarchy
  – Non-root, non-leaf nodes behave as both coordinators and subordinates

Presumed Abort/Commit

• Goal: Reduce the number of log-writes and messages

• Presumed abort (PA)
  – Don’t ACK aborts, don’t force-write logs for aborts
  – After failure and restart, ask the coordinator what happened
  – If coordinator doesn’t find any information, it assumes abort
  – Much fewer messages with read-only transactions

• Presumed commit (PC)
  – Don’t ACK commits, don’t force-write logs for commits
  – Some tricky details
  – Better than PA if commit is more common than abort

• Can choose on a per-transaction basis

Miscellaneous

• Summary of results
  – PA always better than 2P
  – If few updating subords, PA is better; if many updating subords, PC is better
  – Choice between PA and PC to be made on per-transaction basis

• 2-Phase commit is a blocking protocol
  – Waits on the coordinator/its parent in the hierarchy
  – May wait for a long time if the coordinator goes down

• 3-Phase commit solves this problem
  – Generally considered too expensive and not used

• 2-Phase Commit in Oracle
3 Dangers of Replication

Replication

- Another important consideration for distributed databases
- Why replication?
  - Better availability
    - Critical in mobile applications, other places where disconnections common
  - Better performance
    - Don’t need to go to the “master” site for every update
    - “Caches” are a form of replica
- Why “not” replication?
  - Keeping the copies up-to-date very tricky

Replication: Issues

- How are copies kept in-sync: Eager vs Lazy
  - Eager: All copies updated in a single transaction
    - Say using two-phase commit
    - Deadlocks can happen
  - Lazy: Updates propagated lazily
    - reconciliations may be required
- Who owns the objects: Group vs Master
  - Group: Everyone owns the object and can update
  - Master: Single owner; must take permission from it to update
- CODA is Group+Lazy
  - Everyone can update their copies, reconciliations needed
  - Same with cvs or svn

Dangers of Replication

- Dangers of Replication and a Solution: Jim Gray et al.; SIGMOD 1996
  - An extensive performance study comparing deadlocks, number of messages etc...
  - Some results:
    - Eager Master: Deadlocks a bit lower
    - Lazy Group: The transactions with deadlocks need to be reconciled now, so $O(n^3)$
    - Lazy Master: Deadlocks grow as $O(n^2)$
    - Etc. . .