Combining Concurrency Control and Recovery

Instructor: Matei Zaharia

cs245.stanford.edu
Outline

What makes a schedule serializable?
Conflict serializability
Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation

Concurrency control + recovery
Concurrent Control & Recovery

Example:

\[
\begin{align*}
&\text{Tj} & &\text{Ti} \\
&\vdots & &\vdots \\
&wj(A) & &ri(A) \\
&\vdots & &\text{Commit Ti} \\
&\vdots & &\vdots \\
&\text{Abort Tj} & &\vdots
\end{align*}
\]

Non-persistent commit (bad!) avoided by recoverable schedules
Example:

Tj

wj(A)

Abort Tj

Ti

ri(A)

wi(B)

[Commit Ti]

avoided by avoids-cascading rollback (ACR) schedules

Cascading rollback (bad!)
Core Problem

Schedule is conflict serializable

\[ T_j \rightarrow T_i \]

But not recoverable
To Resolve This

Need to mark “final” decision for each transaction:

» **Commit decision**: system guarantees transaction will or has completed, no matter what

» **Abort decision**: system guarantees transaction will or has been rolled back
To Model This, 2 New Actions:

c_i = \text{transaction } T_i \text{ commits}

a_i = \text{transaction } T_i \text{ aborts}
Back to Example

\[ T_j \quad T_i \]
\[ \vdots \quad \vdots \]
\[ w_j(A) \quad r_i(A) \]
\[ \vdots \quad \vdots \]

\[ C_i \quad \text{can we commit here?} \]
**Definition**

$T_i$ reads from $T_j$ in $S$ ($T_j \Rightarrow_S T_i$) if:

1. $w_j(A) <_S r_i(A)$

2. $a_j \not<_S r(A)$ ($\not<_S$: does not precede)

3. If $w_j(A) <_S w_k(A) <_S r_i(A)$ then $a_k <_S r_i(A)$
Definition

Schedule S is **recoverable** if

whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$

then $C_j <_S C_i$
Notes

In all transactions, reads and writes must precede commits or aborts

\[ \iff \ \text{If } c_i \in T_i, \text{ then } r_i(A) < a_i, \ w_i(A) < a_i \]

\[ \iff \ \text{If } a_i \in T_i, \text{ then } r_i(A) < a_i, \ w_i(A) < a_i \]

Also, just one of \( c_i, a_i \) per transaction
How to Achieve Recoverable Schedules?
With 2PL, Hold Write Locks Until Commit ("Strict 2PL")

<table>
<thead>
<tr>
<th></th>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wj(A)</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Cj</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>uj(A)</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>ri(A)</td>
<td>:</td>
</tr>
</tbody>
</table>
With Validation, No Change!

Each transaction’s validation point is its commit point, and only write after
Definitions

S is **recoverable** if each transaction commits only after all transactions from which it read have committed.

S **avoids cascading rollback** if each transaction may read only those values written by committed transactions.

S is **strict** if each transaction may read and write only items previously written by committed transactions (≡ strict 2PL).
Relationship of Recoverable, ACR & Strict Schedules

Diagram:
- Recoverable
  - ACR
    - Strict
      - Serial
Examples

Recoverable:

\[ w_1(A) \ w_1(B) \ w_2(A) \ r_2(B) \ c_1 \ c_2 \]

Avoids Cascading Rollback:

\[ w_1(A) \ w_1(B) \ w_2(A) \ c_1 \ r_2(B) \ c_2 \]

Strict:

\[ w_1(A) \ w_1(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2 \]
Recoverability & Serializability

Every strict schedule is serializable

**Proof:** equivalent to serial schedule based on the order of commit points

» Only read/write from previously committed transactions
Recoverability & Serializability

- Serializable
- Serial
- Strict
- ACR
- Recoverable
Distributed Databases

Instructor: Matei Zaharia

cs245.stanford.edu
Why Distribute Our DB?

Store the same data item on multiple nodes to survive node failures (replication)

Divide data items & work across nodes to increase scale, performance (partitioning)

Related reasons:
» Maintenance without downtime
» Elastic resource use (don’t pay when unused)
Outline

Replication strategies
Partitioning strategies
AC & 2PC
CAP
Avoiding coordination
Outline

Replication strategies

Partitioning strategies

AC & 2PC

CAP

Avoiding coordination
Replication

General problem:
» How do recover from server failures?
» How to handle network failures?
The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz
Replication

Store each data item on multiple nodes!

Question: how to read/write to them?
Primary-Backup

Elect one node “primary”

Store other copies on “backup”

Send requests to primary, which then forwards operations or logs to backups

Backup coordination is either:
  » Synchronous (write to backups before acking)
  » Asynchronous (backups slightly stale)
Quorum Replication

Read and write to intersecting sets of servers; no one “primary”

Common: majority quorum
   » More exotic ones exist, like grid quorums

Surprise: primary-backup is a quorum too!
What If We Don’t Have Intersection?
What If We Don’t Have Intersection?

Alternative: “eventual consistency”
  » If writes stop, eventually all replicas will contain the same data
  » Basic idea: asynchronously broadcast all writes to all replicas

When is this acceptable?
How Many Replicas?

In general, to survive $F$ fail-stop failures, need $F+1$ replicas

Question: what if replicas fail arbitrarily? Adversarially?
What To Do During Failures?

Cannot contact primary?
What To Do During Failures?

Cannot contact primary?
  » Is the primary failed?
  » Or can we simply not contact it?
What To Do During Failures?

Cannot contact majority?
   » Is the majority failed?
   » Or can we simply not contact it?
Solution to Failures:

Traditional DB: page the DBA

Distributed computing: use consensus

» Several algorithms: Paxos, Raft

» Today: many implementations
  • Zookeeper, etcd, Consul

» Idea: keep a reliable, distributed shared record of who is “primary”
Consensus in a Nutshell

Goal: distributed agreement
   » e.g., on who is primary

Participants broadcast votes
   » If majority of notes ever accept a vote $v$, then they will eventually choose $v$
   » In the event of failures, retry
   » Randomization greatly helps!

Take CS244B
What To Do During Failures?

Cannot contact majority?
  » Is the majority failed?
  » Or can we simply not contact it?

Consensus can provide an answer!
  » Although we may need to stall…
  » (more on that later)
Replication Summary

Store each data item on multiple nodes!

Question: how to read/write to them?
» Answers: primary-backup, quorums
» Use consensus to decide on configuration
Outline

Replication strategies

Partitioning strategies

AC & 2PC

CAP

Avoiding coordination
Partitioning

General problem:
  » Databases are big!
  » What if we don’t want to store the whole database on each server?
Partitioning Basics

Split database into chunks called “partitions”
  » Typically partition by row
  » Can also partition by column (rare)

Put one or more partitions per server
Partitioning Strategies

Hash keys to servers
  » Random assignment

Partition keys by range
  » Keys stored contiguously

What if servers fail (or we add servers)?
  » Rebalance partitions (use consensus!)

Pros/cons of hash vs range partitioning?
What About Distributed Transactions?

Replication:
» Must make sure replicas stay up to date
» Need to reliably replicate commit log!

Partitioning:
» Must make sure all partitions commit/abort
» Need cross-partition concurrency control!
Outline

Replication strategies

Partitioning strategies

AC & 2PC

CAP

Avoiding coordination
Atomic Commitment

Informally: either all participants commit a transaction, or none do

“participants” = partitions involved in a given transaction
So, What’s Hard?
So, What’s Hard?

All the problems as consensus...

...plus, if any node votes to abort, all must decide to abort

  » In consensus, simply need agreement on “some” value
Two-Phase Commit

Canonical protocol for atomic commitment (developed 1976-1978)

Basis for most fancier protocols

Widely used in practice

Use a transaction *coordinator*
  » Usually client – not always!
Two Phase Commit (2PC)

1. Transaction coordinator sends *prepare* message to each participating node

2. Each participating node responds to coordinator with *prepared* or *no*

3. If coordinator receives all *prepared*:
   » Broadcast *commit*

4. If coordinator receives any *no*:
   » Broadcast *abort*
Case 1: Commit

Coordinator

Request-to-Prepare

Prepared

Commit

Done

Participant
Case 2: Abort

Coordinator

Request-to-Prepare

No

Abort

Done

Participant
2PC + Validation

Participants perform validation upon receipt of *prepare* message

Validation essentially blocks between *prepare* and *commit* message
2PC + 2PL

Traditionally: run 2PC at commit time
  » i.e., perform locking as usual, then run 2PC when transaction would normally commit

Under strict 2PL, run 2PC before unlocking write locks
2PC + Logging

Log records must be flushed to disk on each participant before it replies to prepare

» (And updates must be replicated to F other replicas if doing replication)