

CSE 444: Database Internals

Lecture 24 Two-Phase Commit (2PC)

CSE 444 - Spring 2019

1

References

- Ullman book: Section 20.5
- Ramakrishnan book: Chapter 22

CSE 444 - Spring 2019

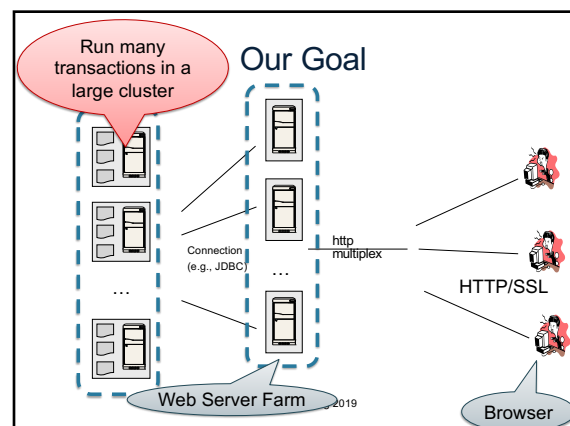
4

We are Learning about Scaling DBMSs

- **Scaling the execution of a query**
 - Parallel DBMS
 - MapReduce
 - Spark
- **Scaling transactions**
 - Distributed transactions
 - Replication
 - Scaling with NoSQL and NewSQL

CSE 444 - Spring 2019

5



Transaction Scaling Challenges

- **Distribution**
 - There is a limit on transactions/sec on one server
 - Need to partition the database across multiple servers
 - If a transaction touches one machine, life is good!
 - If a transaction touches multiple machines, ACID becomes extremely expensive! Need two-phase commit
- **Replication**
 - Replication can help to increase throughput and lower latency
 - Create multiple copies of each database partition
 - Spread queries across these replicas
 - Easy for reads but writes, once again, become expensive!

CSE 444 - Spring 2019

7

Distributed Transactions

- **Concurrency control**
- **Failure recovery**
 - Transaction must be committed at all sites or at none of the sites!
 - No matter what failures occur and when they occur
 - **Two-phase commit protocol (2PC)**

CSE 444 - Spring 2019

8

Distributed Concurrency Control

- In theory, different techniques are possible
 - Pessimistic, optimistic, locking, timestamps
- In practice, distributed two-phase locking
 - Simultaneously hold locks at all sites involved
- Deadlock detection techniques
 - Global wait-for graph (not very practical)
 - Timeouts
- If deadlock: abort least costly local transaction

CSE 444 - Spring 2019 9

- CSE 444 - Spring 2019

9

Two-Phase Commit: Motivation

Coordinator

1) User decides to commit

2) COMMIT

Subordinate 1

3) COMMIT

4) Coordinator crashes

What do we do now?

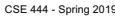
Subordinate 2

Subordinate 3

But I already aborted!

CSE 444 - Spring 2019

10



10

Two-Phase Commit Protocol

- One coordinator and many subordinates
 - **Phase 1: prepare**
 - All subordinates must flush tail of write-ahead log to disk before ack
 - Must ensure that if coordinator decides to commit, they can commit!
 - **Phase 2: commit or abort**
 - Log records for 2PC include transaction and coordinator ids
 - Coordinator also logs ids of all subordinates
- **Principle**
 - Whenever a process makes a decision: vote yes/no or commit/abort
 - Or whenever a subordinate wants to respond to a message: ack
 - **First force-write a log record** (to make sure it survives a failure)
 - **Only then send message about decision**

CSE 444 - Spring 2019 11

- CSE 444 - Spring 2019

11

2PC: Phase 1, Prepare

The diagram illustrates the 'Prepare' phase of a Two-Phase Commit (2PC) protocol. It features a Coordinator (orange circle) and three Subordinates (grey circles). The process is as follows:

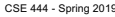
- 1) User decides to commit:** This event triggers the Coordinator to initiate the process.
- 2) PREPARE:** The Coordinator sends this message to all three Subordinates.
- 3) Force-write: prepare:** Each Subordinate performs this action locally.
- 4) YES:** All three Subordinates respond with 'YES' to the Coordinator.

Arrows indicate the flow of messages from the Coordinator to the Subordinates and back. The steps are color-coded: blue for user/initiation, green for protocol messages, and purple for local actions.

```
graph TD;
    User((User)) -- "1) User decides to commit" --> Coord((Coordinator));
    Coord -- "2) PREPARE" --> S1((Subordinate 1));
    Coord -- "2) PREPARE" --> S2((Subordinate 2));
    Coord -- "2) PREPARE" --> S3((Subordinate 3));
    S1 -- "4) YES" --> Coord;
    S2 -- "4) YES" --> Coord;
    S3 -- "4) YES" --> Coord;
    S1 -- "3) Force-write: prepare" --> S1;
    S2 -- "3) Force-write: prepare" --> S2;
    S3 -- "3) Force-write: prepare" --> S3;
```

CSE 444 - Spring 2019

12



12

[illegible]

13

2PC with Abort

```
graph TD; C((Coordinator)); S1((Subordinate 1)); S2((Subordinate 2)); S3((Subordinate 3)); C -- "2) PREPARE" --> S1; C -- "2) PREPARE" --> S2; C -- "2) PREPARE" --> S3; S1 -- "4) YES" --> C; S2 -- "4) No" --> C; S3 -- "4) NO" --> C; C -- "3) Force-write: abort" --> S1; C -- "3) Force-write: abort" --> S2; C -- "3) Force-write: abort" --> S3; C -- "5) Abort transaction and 'forget' it" --> S1; C -- "5) Abort transaction and 'forget' it" --> S2; C -- "5) Abort transaction and 'forget' it" --> S3;
```

1) User decides to commit

2) PREPARE

3) Force-write: prepare

4) YES

4) No

4) NO

2) PREPARE

3) Force-write: abort

5) Abort transaction and "forget" it

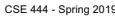
3) Force-write: abort

5) Abort transaction and "forget" it

5) Abort transaction and "forget" it

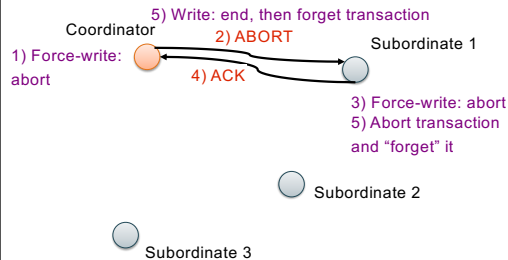
CSE 444 - Spring 2019

14



14

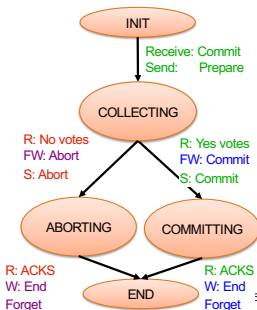
2PC with Abort



15

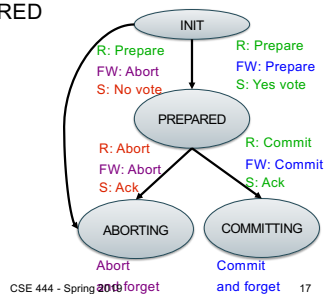
Coordinator State Machine

- All states involve waiting for messages



Subordinate State Machine

- INIT and PREPARED involve waiting



CSE 444 - Spring 2019

17

Handling Site Failures

- Approach 1: no site failure detection
 - Can only do retrying & blocking
- Approach 2: timeouts
 - Since unilateral abort is ok,
 - Subordinate can timeout in init state
 - Coordinator can timeout in collecting state
 - Prepared state is still blocking
- 2PC is a blocking protocol

CSE 444 - Spring 2019

18

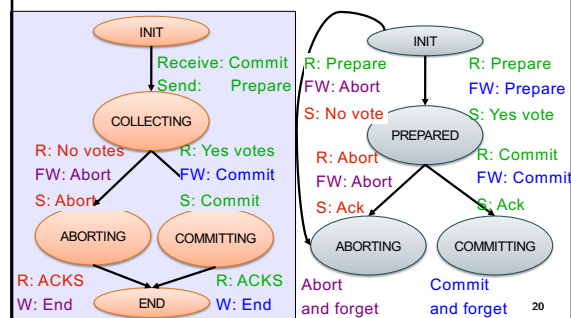
Site Failure Handling Principles

- Retry mechanism
 - In prepared state, periodically query coordinator
 - In committing/aborting state, periodically resend messages to subordinates
- If doesn't know anything about transaction respond "abort" to inquiry messages about fate of transaction
- If there are no log records for a transaction after a crash then abort transaction and "forget" it

CSE 444 - Spring 2019

19

Site Failure Scenarios



20

Observations

- Coordinator keeps transaction in transactions table until it receives all acks
 - To ensure subordinates know to commit or abort
 - So acks enable coordinator to "forget" about transaction
- After crash, if recovery process finds no log records for a transaction, the transaction is presumed to have aborted
- Read-only subtransactions: no changes ever need to be undone nor redone

CSE 444 - Spring 2019

21

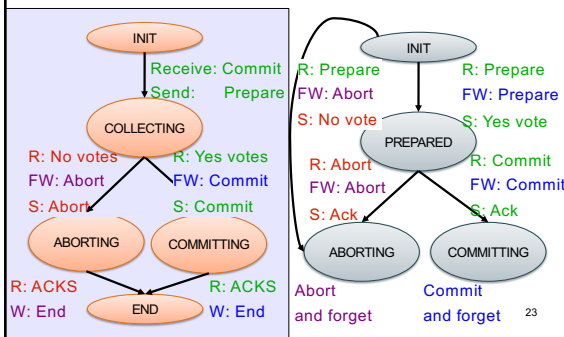
Presumed Abort Protocol

- Optimization goals
 - Fewer messages and fewer force-writes
- Principle
 - If nothing known about a transaction, assume ABORT
- Aborting transactions need no force-writing
- Avoid log records for read-only transactions
 - Reply with a READ vote instead of YES vote
- Optimizes read-only transactions

CSE 444 - Spring 2019

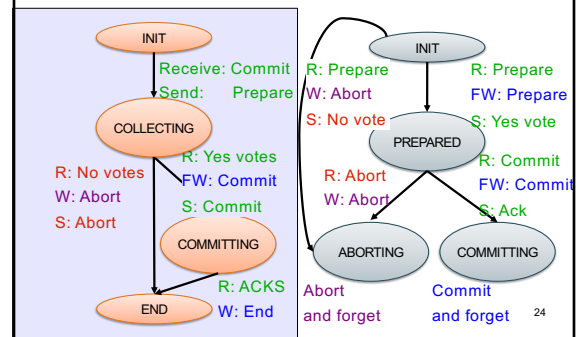
22

2PC State Machines (repeat)



23

Presumed Abort State Machines



24