Lecture 18 : Distributed Transactions **11/12/2014**
Aaron Elmore Lecturing

R*: System R distributed follow-on. Very influential.

Today: Two-phase commit.

What's the purpose of two-phase commit?

(Distributed Transactions)

Transaction
BEGIN
INSERT a ---- > S1
INSERT b ---- > S2
INSERT c ---- > S3
COMMIT

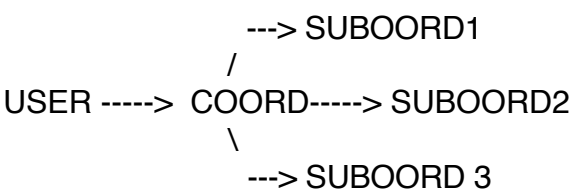
Why doesn't existing commit protocol work?

Suppose S1 crashes, but S2 and S3 succeed. S2 and S3 shouldn't commit unless S1 commits.

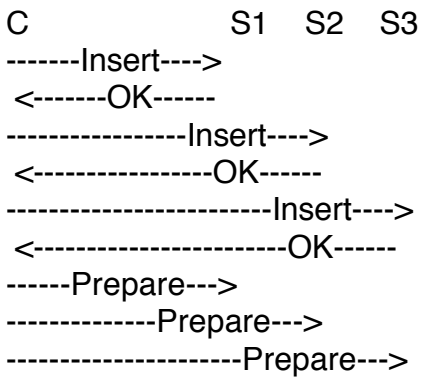
Also want nested xactions

```
BEGIN
READ A
INSERT B
      BEGIN
            READ C
            INSERT A
      END
END
```

What's the architecture here?



- Query statements arrive at COORD.
- COORD sends statements to SUBOORD.



- Once COORD gets a COMMIT statement from USER, it initiates 2PC.

- Each site has a *recovery process* that keeps track of the fate of transactions running on the site, contacts and responds to contacts from remote sites re: running transactions.

Why do we need such a complicated protocol?
Why not just send COMMIT messages to all sites once they've finished their share of the work?

One of them might fail during COMMIT processing, which would require us to be able to ABORT suboords that have already committed.

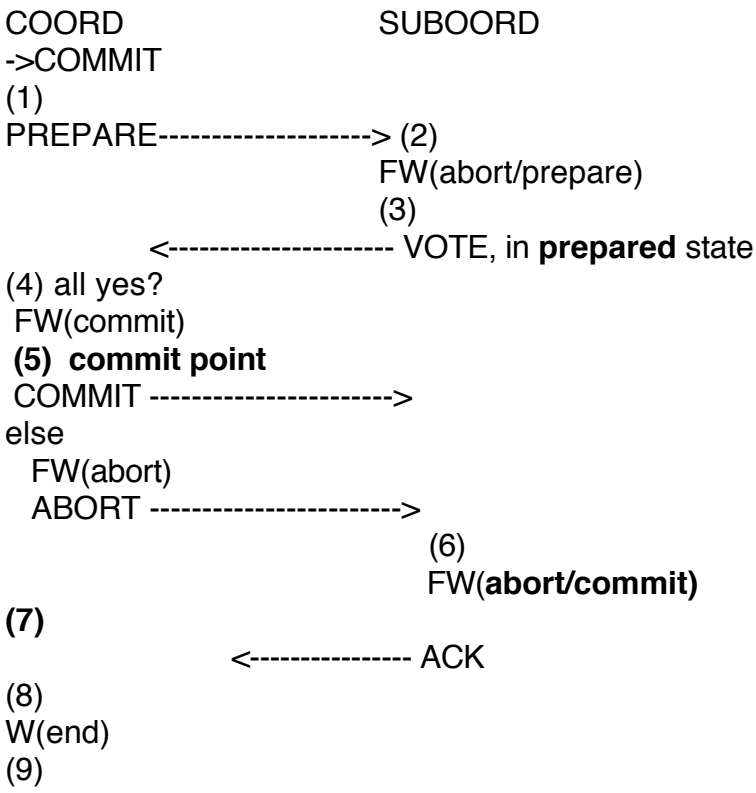
So how does 2PC avoid this?

Via PREPARE.

What does a site being PREPARED mean?

If a site is prepared, it can COMMIT the xaction, even if they crash before getting the COMMIT message. So if all sites are PREPARED, it's definitely OK to commit.

Ok -- let's look at the protocol -- requires careful interleaving of logging and messaging.



What do log records look like?

all records - tid, coordid
prepare records -- list of locks held
coord commit/abort -- list of suboords

What's the deal with force writes?
log up to that point must be on disk before you can continue.

Let's look at what happens in the face of various failures.
(1) Transaction aborted

Coord -- will recover, abort transaction just as in normal recovery
(discarding all state)
Suboord -- must timeout eventually, once it contacts *coord*, which has no record of xaction, it will abort. (Coord in basic protocol replies abort in no information case)

(2) Abort

Coord -- will never hear reply, will abort (how does it tell the suboord failed?)
Suboord -- will recover, rollback xaction during recovery

(3) Is in **prepared** state. Need to contact COORD to determine fate. Couple of options:

- It already sent its vote, and coord is waiting for SUBOORD to send an ack -- thus, SUBOORD can learn fate.

- It didn't send its vote, in which case COORD may or may not have timed out. If it hasn't timed out, it can vote. If COORD has timed out, it *must* have aborted, and will tell the SUBOORD this.

(4) Crashed before receiving all votes. Abort.
COORD aborts during recovery
SUBOORD eventually times out, when it contacts COORD it will learn xaction aborted.

(5) Crashed after writing commit record. Commit.

COORD recovers into **committing** state. Must send commit messages and collect ACKs from all sites.

Couple of possibilities:

SUBOORD may have already written commit message/sent ack, and forgotten about xaction. In which case, it should just ACK and do nothing.

SUBOORD may not have written commit message, and is waiting, in which case, this allows it to go forward.

Note that SUBOORD cannot time out xaction at this point -- it must wait to hear from COORD before committing/aborting.

(6) Crashed before receiving COMMIT/abort

Upon recovery, SUBOORD contacts COORD, asks about fate of xaction. COORD cannot forget state since it has not heard an ack yet.

(7) Crashed after writing COMMIT record, before ACKing.

SUBOORD will recover, transaction will be committed. COORD will periodically send a COMMIT message, which SUBOORD will ACK without writing any additional state.

(8) COORD Crashed after receiving some ACKs.

COORD will send COMMIT/ABORT to all SUBOORDS, who will ACK.

(9) Coord crashes after writing END.

Nothing needs to be done.

What happens if we send a VOTE before writing the PREPARED record?

Trouble! SUBOORD might recover and rollback, when it should have committed.

What happens if SUBOORD sends and ACK before writing the COMMIT record?

SUBOORD might recover, contact COORD, which will know nothing about xaction (because it wrote the END record), and reply "ABORTED", which would be wrong.

What if COORD replied "committed" by default? Problem in step 4.

Read only sites.

If a SUBOORD is read only, it can send a "READ VOTE". It doesn't need to write any log records, and can forget the transaction after it votes.

COORD doesn't need to send ABORT/COMMIT messages to READ only sites.

If all sites are read only, no ABORT/COMMIT messages need to be sent.

Presumed Abort

Notice that in the absence of information, we abort. This means we don't need to:
- Force write abort record on any site.
- Send ABORT messages at all (though we still may want to for efficiency reasons)
- Send/wait for ACK messages for aborts
- Write END record for aborted transactions on COORD.

Presumed Commit

Change protocol so that we return "COMMIT" when there is no information about a transaction. This means we do not have to acknowledge COMMIT messages, and we don't have to force write COMMIT records or END records for COMMITTED transactions.

What does this break?

Suppose COORD crashes after having received some but not all votes. It will then tell all prepared SUBOORDs that the transaction COMMITed when they inquire, but this may not be correct, since it's not safe to assume that all SUBOORDs are able to commit.

Soln?

Force write a list of SUBOORDs at COORD before sending PREPARE message. Then COORD can figure out who was involved in transaction and tell them to ABORT (or figure out what they wanted to vote and decide if transaction can go forward.)

Messages for committing transaction

	Coord U	Suboord U	 R
Standard	2W,1F,2M,1M(R),2M(W) 2	W,2F,2M	0W,0F,1M
PA	2W,1F,1M(R),2M(W)	2W,2F,2M	0W,0F,1M
PC	2W,2F,1M(R),2M(W)	2W,1F,1M	0W,0F,1M

Deadlock detection:

What is the problem with deadlocks in a parallel DB?
Two sites can serialize transactions in different orders, be waiting on each other to commit:

T1S1	T2S1	T1S2	T2S2
locked: A	waiting: A	waiting: B	locked: B

What's the problem? S2 has ordered T1 after T2, S1 has ordered T1 before T2

This is a deadlock.

What do we do to fix it?

Build a global waits-for graph by sending local waits-for information to other cites.

T1 ------ WF -------> T2
<------ WF ------

Detect cycles.
Hard because sites are running asynchronously. Need to ensure that we don't "accidentally" detect a deadlock by using stale waits-for info.

Rather than sending information to everyone else, can limit the number of sites that

waits-for information goes to:

For example:
S1: T1 < ---- T2
S2: T1 -----> T2

Here, S2 sends info to S1 because T1 waits for T2 and T1 < T2. S1 does not send to S2. Then, S1 detects deadlock.

What victim to choose?

Just pick locally.

What do real databases do?

Apparently, they used presumed abort, and detect deadlocks via timeout, not deadlock detection.

Why presumed abort?

Not clear -- not many transactions abort. PC does add an extra log record. Maybe it's just simpler and they haven't bothered to do PC.

Why timeout based DDD?

Complexity

What if we want two sites to commit the transaction at exactly the same time?

Two generals paradox! Can't guarantee that you can get consensus in bounded time in the face of a lossy network.

Barack Hilary
Attack at dawn!------>
<-------------------------OK
Let's roll!---------->
<----------------------Got it.

R can't know that B heard his last message. Therefore, R may not have heard B's last message, and so on....

If there is a non-zero probability of delivery, retry's will eventually guarantee success, but not in a bounded time.