Lecture 17 : Distributed Transactions **4/26/2021**

Today: Two-phase commit.

Last time: Parallel query processing

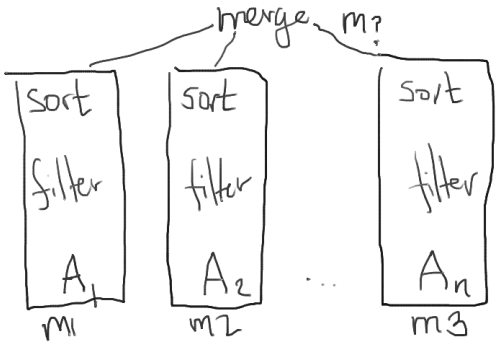
Recap:

Main ways to get parallelism:
Across queries:
- run multiple queries simultaneously

With in a query:
- run groups of operators in separate threads ("pipeline")

- partition data across multiple processors / threads ("partition")

Partitioning is the main way databases parallelize one query -- only way to get high levels of speedup because of limited pipeline length and frequency of blocking operators.

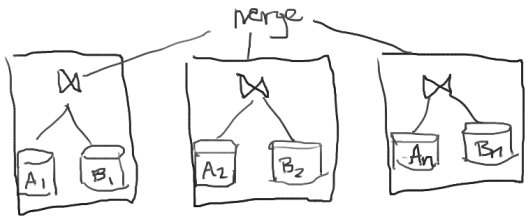


Three main types of partitioning: hash, range and round robin.

Talked about join algorithms; several options:
0) If tables are partitioned properly, nothing to do
1) Copy smaller table
2) Re-partition one or both tables (depending on initial partitioning)

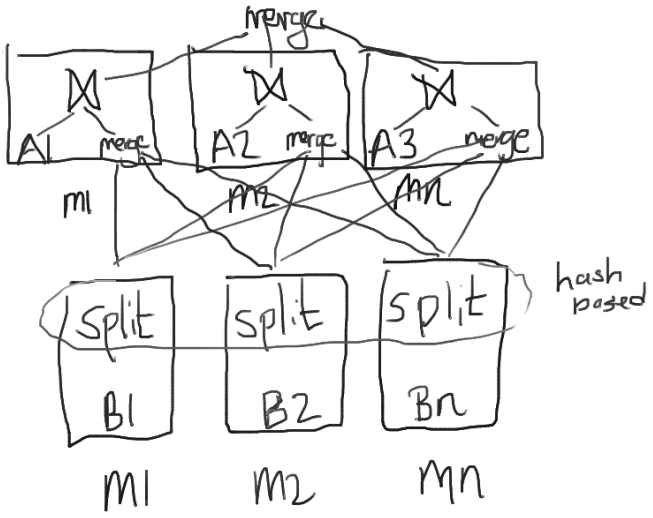
Ex: Join A.a = B.b, where A partitioned on a and B partitioned on b. No repartitioning work to do!

SELECT * FROM A,B WHERE A.a = B.b
n machines, hash fxn H
Hash A on a, H(a) -> 1..n
Hash B on b, B(b) -> 1..n



Ex-- Join A.a = B.b, where A partitioned on A.a

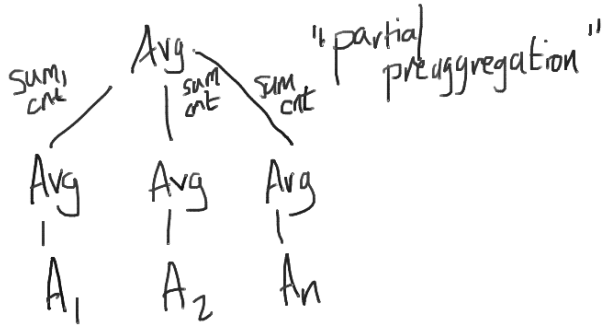
Hash B on c, H(c) -> 1..n



Generalizes to the case when both tables have to be repartitioned.

Aggregation: Can run aggregates in parallel, then merge groups.
Example:

select avg(f) from A



Standard way of expressing aggregates:

INIT
MERGE -- associative & commutative
FINAL

Ok -- so now lets talk about distributed transaction processing. Primary way we achieve this is **two phase commit.**

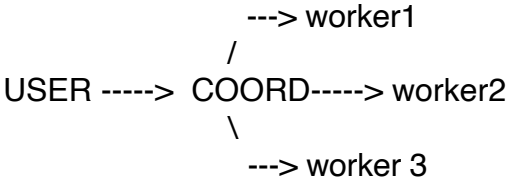
What's the purpose of two-phase commit?
(Distributed transactions, on multiple machines, where machines can fail independently.)

Transaction
BEGIN
INSERT a ---- > worker1
INSERT b ---- > worker2
INSERT c ---- > worker3
COMMIT

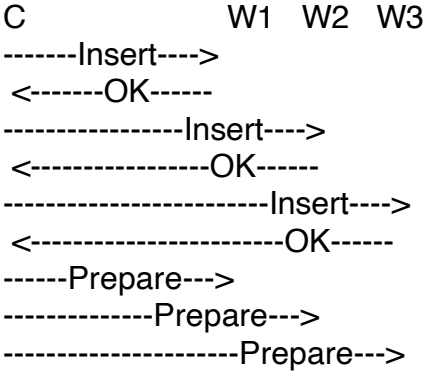
Why doesn't existing commit protocol work?

Suppose W1 crashes, but W2 and W2 succeed. S2 and S3 shouldn't commit unless S1 commits.

What's the architecture here?



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



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

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 workers that have already committed.

So how does 2PC avoid this?

Via PREPARE.

What does a site being PREPARED mean?

If a worker site is prepared, it is ready and able to COMMIT the xaction. It must retain the ability to do this even if it crashes. If all sites are PREPARED, the coordinator can unilaterally decided to abort or commit. (Requires logging!)

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

Preliminaries:

- 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.

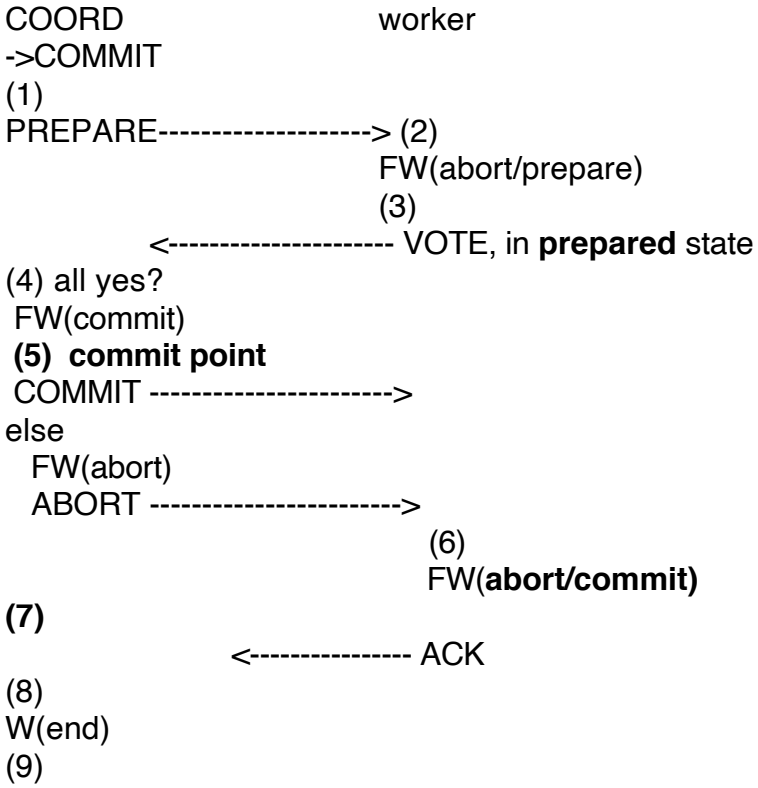
- If a site is prepared and crashes, it needs to ask coordinator about the outcome of the transaction on recovery.

- So when can coordinator / worker forget about state of the transaction? Coordinator needs to make sure all workers have learned about xaction state, and workers need to make sure coordinator has learned about their "vote".

What is in log records?

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

Protocol example:



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)
worker -- 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 worker failed?)
worker -- 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 worker to send an ack -- thus, worker 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 worker this.

(4) Crashed before receiving all votes. Abort.
COORD aborts during recovery
worker 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:

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

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

Note that worker 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, worker 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.

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

(8) COORD Crashed after receiving some ACKs.

COORD will send COMMIT/ABORT to all workerS, 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! worker might recover and rollback, when it should have committed.

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

worker 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 worker 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. Will ack any commits sent by COORD.

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. Changes:
- workers don't acknowledge COMMIT messages
- workers don't have force write COMMIT (still have to force on COORD).
- Don't write END records for COMMITs on COORD.

What does this break?

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

Soln?

Force write a list of workers at COORD before sending PREPARE message (a "COLLECTING" message). Then COORD can figure out who was involved in transaction and tell them to ABORT.

Messages for committing transaction

	Coord Update or Readonly	worker Update	Read-Only
Standard	2W,1F,1M(R/O),2M(U)	2W,2F,2M	0W,0F,1M
PA	2W,1F,1M(R/O),2M(U)	2W,2F,2M	0W,0F,1M
PC	2W,2F,1M(R/O),2M(U)	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:

T1W1	T2W1	T1W2	T2W2
locked: A	waiting: A	waiting: B	locked: B

What's the problem? W2 has ordered T1 after T2, W1 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:
W1: T1 < ---- T2
W2: T1 -----> T2

Here, W2 sends info to W1 because T1 waits for T2 and T1 < T2. W1 does not send to W2. Then, W1 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.