last time we were talking about parallel DBs
partitioned data across multiple servers
we mostly discussed read-only queries
what about read/write queries?

high-level model
a bunch of servers
rows are partitioned over servers
each server runs a complete DB on its partition
SQL, locks, logging
external client connects to one server: Transaction Coordinator (TC)
sends it commands for whole system
TC farms them out to correct "subordinate" server
TC collects results, returns them to client
TC and servers exchange messages over a LAN

example transaction:
begin
SELECT A ...
SELECT B ...
UPDATE A ...
UPDATE B ...
commit

diagram
A on S1, B on S2
client connects to S3
S3 sends SELECT A to S1, gets result
S3 sends SELECT B to S2, gets result
S3 sends UPDATE A to S1
S3 sends UPDATE B to S2
S3 sends "transaction completed" reply to client
but wait, this is not enough!

what about locking?
each r/w acquires lock on server w/ the data
so: S1/A/S, S2/B/S, S1/A/S, S1/A/X

when should the system release the locks?
remember we want strict two-phase locking
for serializability and no cascading aborts
so can't release until after commit
so there must be at least one more message:
TC tells S1,S2 that the transaction is over

can we get deadlock?
yes, for example if we run two of this transaction
in general a subordinate could block+deadlock at any read or write
let's assume a global deadlock detector
which notifies a transaction that it must abort
so a subordinate might be aborted at any step

more generally, a subordinate can fail for a number of reasons
deadlock
integrity check (e.g. insert but primary key not unique)
crash
network failure

what if one subordinate fails before completing its update?
and the other subordinate didn't fail?
we want atomic transactions!
so TC must detect this situation, tell the other subordinate to abort+UNDO

we need an "atomic commitment" protocol
all subordinates complete their tasks
or none

Two-Phase Commit is the standard atomic commitment protocol

2PC message flow for ordinary operation

```
Client  TC  Subordinate
------------
-- SQL cmds -->
  acquire locks
  if update,
    append to log
  update blocks
  check deadlock, integrity, &c
-- PREPARE -->
  [log prepare or abort]
  <!-- VOTE YES/NO --
    wait for all VOTEs
  [log com/ab]
  <!-- OK/NOT OK ----
  --> COMMIT/ABORT -->
  [log commit or abort]
    release locks
  <!-- ACK ------
    wait for all ACKs
  [log end]
```

notes for ordinary operation:
if subordinate voted YES
  doesn't know outcome until COMMIT/ABORT msg from TC
  since some other subordinate might vote NO
  thus must be prepared to do either
  must also hold locks
if subordinate voted NO
  for sure whole xaction will be aborted
  so subordinate can immediately undo and release locks

what if TC gets no response to a PREPARE?
  net failure, or subordinate crashed
  TC keeps sending PREPARE for a while
what should TC do if still no response?
  abort, and send ABORT msgs
  why is this safe?
  at this point, TC has sent no COMMITs
  all subordinates waiting for PREPARE or COMMIT
  so no partial updates have been made visible
if subordinate crashed and restarted
  can't resume a process after crash
  can roll back, using log
  respond VOTE NO to TC's PREPARE

what if TC gets no response to a COMMIT?
  re-send for a while
what should TC do if still no response?
  can TC decide to abort instead?
  no: other subordinates may have received COMMIT, released locks
  so the TC must wait
what should subordinate do when it restarts and runs recovery?
  no commit record: should it therefore roll back?
  subordinate must distinguish crash before PREPARE from crash after
so must write a PREPARE log record to disk before sending VOTE YES
then during recovery:
  if PREPARE record in log, ask TC whether to commit
  if no PREPARE record, roll back

PREPARE log record must list the locks held
  since must re-acquire while waiting to hear from TC

implication:
  TC must remember whether transaction aborted or committed
  when can TC forget about a transaction?
    after it hears all ACKs
    it then knows all subordinates know the outcome

what if subordinate never gets a PREPARE?
  TC crashed, or network failure
  subordinate can abort w/o asking TC

what if subordinate sent VOTE YES, got no COMMIT/ABORT?
  cannot unilaterally abort
  must wait for TC, ask it what happened

what should TC do during recovery if it crashes?
  and what should it tell inquiring subordinates?
    if TC could not have sent a commit
      it can just abort, reply "abort" if anyone inquires
    if TC might have sent a commit
      it cannot change its mind, since one subordinate may have released locks

how can the TC tell on recovery if it might have sent a commit msg?
  it must log a commit record to disk after VOTEs collected, before sending COMMIT
  on recovery, look in log:
    if no commit record, can abort
    if commit record, must answer "commit" to any subordinate queries

when can the TC forget about a transaction?
  when no subordinate could possibly inquire
  so TC keeps track of who has ACKed a COMMIT/ABORT msg
    ACK implies subordinate has logged a commit/abort record
  when all have ACKed, can forget
    delete from memory
    can GC that part of log
  paper's "end" log record tells recovery not to bother

concerns
  2pc can block: subordinate may have to wait forever if TC down
  while holding locks!
    when can we resolve?
      subordinate can always abort if hasn't replied to PREPARE
    when do we have to wait?
      TC crashes after last PREPARE sent, before first COMMIT sent
      did it time out and abort locally and say NO to client??
      did it commit locally and say YES to client?
  want to limit window of vulnerability to TC crashes
    as little time as possible when subordinates can't unilaterally abort
    this one reason for separate PREPARE at very end
      rather than yes/no replies to each action RPC
  performance: 2pc adds burdens to TC, subordinates
    log forces -- super painful
    messages -- somewhat painful
    TC must keep state -- somewhat painful

what are the forced log records?
(required to recover state after crash)
subordinate prepare / abort
TC commit / abort
subordinate commit / abort

can we get rid of any of these log forces?
TC recovery never really looks at abort records
so there is no point in writing them at all
if didn't force, would be like crash before sending PREPARE, -> abort
subordinates need not force abort log records
if crash, lack of commit and abort and prepare -> can abort unilaterally

Presumed Abort protocol exploits these ideas
YES votes and TC commit work as before
TC abort:
  don't log anything!
  forget about xaction
  send ABORT msgs (so subordinates can release locks)
  don't bother collecting ACK msgs
TC recovery:
  nothing in log for aborted xaction
TC response to queries:
  if no record of xaction, reply "aborted" (hence the "presumed abort" name)
subordinate NO vote:
  don't force abort to log
  release locks

why might we care about Presumed Abort?
  after all, abort is much less common than commit

we can speed up read-only xaction *commit* if we use PA
  don't in general know in advance that an xaction will be r/o
TC sends out PREPAREs
subordinates send READ-ONLY VOTE if could commit but read-only
  if TC gets all READ-ONLYs
    send COMMITs so subordinates can release locks
    TC forgets about xaction w/o logging anything
    subordinates need not log anything either
    that is, convert read-only commit to abort
  if sub missed COMMIT msg, asks what happened, can it release locks
    TC sees no record, says "abort", which is fine

if we weren't using PA, TC would have to force an explicit commit or
  abort log record, and would see no performance win for r/o xactions

how many forced log writes does TC make for committed transaction?
  2PL r/w: 1
  2PL r/o: 1
  PA r/w: 1
  PA r/o: 0

can we have Presumed Commit?
to speed up common case of r/w committed transactions?
  if TC commits: don't force the commit log record to disk
  if TC aborts: force the abort log record to disk
  if TC crash+recover, subordinate asks whether an xaction committed,
    TC replies "commit" if no record of xaction

problem:
  TC sends PREPAREs, gets some YES votes, crashes before seeing all, recovers
  TC recovery sees no commit record, no abort record
  PREPAREd subordinate asks TC if xaction committed
  TC answers "commit", since no record of this xaction
oops, since some other subordinate might have voted NO but
TC crashed before seeing its vote

fix to make Presumed Commit work
TC, before sending PREPARE msgs, logs PREPARE w/ list of subordinates
crash recovery at TC restarts PREPARE processing

summary of costs, counted as # forced writes at TC:

<table>
<thead>
<tr>
<th></th>
<th>TC r/w</th>
<th>TC r/o</th>
<th>SUB r/w</th>
<th>SUB r/o</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PA</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

XXX why does paper say PC requires forced commit at TC?
maybe TC got all yes votes, said "yes" to client, then crashed
after restart, can't contact one of subordinates
changes its mind to "abort"
XXX why does sub do a forced log write for PC?
so it doesn't change its mind about its PREPARE vote?

when do systems use 2pc?
in a single machine room, for parallel DBs
could use to get atomicity across heterogeneous DBs
probably not done very much
many DBs have 2pc interfaces (separate prepare and commit),
but not very standard
could in principle use between different organizations
expedia, one log on united, on one AA
but not done in practice
WAN msgs too slow
don't want expedia's flaky TC to cause locks to be held at United!