Distributed Databases: Concurrency Control

Prof. Chris Clifton
16 April 2021

Distributed Transaction: Locking
Distributed Locking

- All locks are local!
  - Checking for locks happens at each site
  - Wait happens at each site
  - No need to communicate
- 2-phase locking is global
  - No transaction component can release a lock until all are done obtaining the lock
- We get serializability!

Distributed Concurrency Control: Challenges

- Deadlock detection/prevention: Deadlocks can be distributed
  - \( T_1 \) waiting for \( T_2 \) at site A
  - \( T_2 \) waiting for \( T_1 \) at site B
- Replication
  - What if we want to have multiple copies of the data?
Replicated Data

• Thus far, we have assumed that there is only a single copy of each data item.
• This copy is placed at one of the sites, which is responsible for concurrency control and recovery for that data item.
• However, for a data item that is accessed often from different sites, this could lead to a significant amount of communication.
• Moreover, when a site fails, all data residing on that site becomes unavailable.

Replication

• To increase availability of data, and to reduce communication for remote data, data can be replicated.
• From the user’s point of view, replication (like distribution, physical and logical organization of data), should be transparent.
• I.e. the user should not be aware that some (or all) data items are replicated, and should see no difference in performance.
• The user can be a programmer or an end user.
1 Copy Serializability

- The correctness definition for replicated databases is therefore that it should behave as though all transactions are executed in a **serial manner on a single copy database**.
- This is the notion of one copy serializability, i.e. 1SR.
- The user must be given a one copy view of the database.
- How is this achieved?
- Read-only is easy. For writes we must manage carefully!

Write-All approach

- This is the obvious first solution:
  - Reads can be satisfied by any copy in the system,
  - Writes must all modify *every* copy of the data item being written.
- This is a very effective solution – it completely **eliminates the problem** of multiple copies, and gives each txn the correct view.
  - Lock each copy
  - If someone reading a copy, we can’t get write lock
- Very poor in terms of performance and progress:
  - Failures have a [crippling](https://example.com) effect on transactions!
Quorum Consensus Protocol

Quorum consensus protocol for locking

- Each site is assigned a weight; let S be the total of all site weights
- Choose two values read quorum $Q_R$ and write quorum $Q_W$
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
- Each read must lock enough replicas that the sum of the site weights is $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is $\geq Q_w$
- Can choose $Q_r$ and $Q_w$ to tune relative overheads on reads and writes

Write-All-Available

- Allow a transaction to proceed even though failures make it impossible to write all copies of the data.
- Allow the transaction to simply write to every site that is available. Those that are down can be ignored.
- Thus some copies of the data may be out of sync, i.e., may not contain the latest updates.
Example

- Consider the following execution. Note that multiple copies are marked using the upper case subscripts.
  \[ w_0[x_A] w_0[x_B] w_0[y_C] c_0 r_1[y_C] w_1[x_A] c_1 r_2[x_B] w_2[y_C] c_2 \]
- \( T_2 \) reads copy \( x_B \) from \( T_0 \), even though it should have read from \( T_1 \).
- Thus the above history is not equivalent to \( T_0 T_1 T_2 \).
- Is it equivalent to some other serial one-copy history?
  - NO! \( w_0[y_C] < r_1[y_C] < w_2[y_C] \), there is no other equivalent serial execution.
  - This is interesting, because the execution actually seems to be a serial execution of the transactions!!!

Example (contd.)

- So what has gone wrong?
- The problem is that the write by \( T_1 \) into \( x \), did not update all copies of \( x - x_B \) in particular.
- This could only mean that site B must have been down when \( T_1 \) wrote \( x \), and must have recovered before \( T_2 \) read \( x \).
- I.e. the failures must have been as such:
  \[ w_0[x_A] w_0[x_B] w_0[y_C] c_0 r_1[y_C] \text{ fail}_B w_1[x_A] c_1 \text{ Recover}_B r_2[x_B] w_2[y_C] c_2 \]
- Thus the problem is that \( T_2 \) read a copy at a site that had failed and upon recovery did not re-sync with the other sites!
  - Recovery necessary to get concurrency control right!
Handling Failures with Majority Protocol

- The majority protocol with version numbers
  - Each replica of each item has a version number
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
- Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task

Reducing Read Cost

- Quorum consensus can be used to reduce read cost
  - But at increased risk of blocking of writes due to failures
- Use primary copy scheme:
  - perform all updates at primary copy
  - reads only need to be done at primary copy
  - But what if primary copy fails
    - Need to ensure new primary copy is chosen
      - Leases can ensure there is only 1 primary copy at a time
    - New primary copy needs to have latest committed version of data item
      - Can use consensus protocol to avoid blocking
Distributed Deadlock Handling

Timestamp Ordering

- The TM assigns each txn, \( T_i \), a unique timestamp, \( ts(T_i) \).
- No two txns share a timestamp.
- A TO scheduler enforces:
  - **TO Rule**: if \( p_i[x] \) and \( q_j[x] \) are conflicting operations, then the DM processes \( p_i[x] \) before \( q_j[x] \) iff \( ts(T_i) < ts(T_j) \).
Serializability

- **Theorem:** If $H$ is a history representing an execution produced by a TO scheduler, then $H$ is serializable.
- **Proof:** Consider $SG(H)$.
  - If $T_i \rightarrow T_j$ is an edge in $SG(H)$, then there must exist conflicting operations $p_i[x]$ and $q_j[x]$ in $H$ such that $p_i[x] < q_j[x]$.
  - Hence by the TO rule, $ts(T_i) < ts(T_j)$.
  - If there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in $SG(H)$, then by induction, $ts(T_1) < ts(T_1)$!!!

Basic TO

- For each operation, we pass it to the DM as long as it is not too late!
- An operation is too late if a conflicting operation with a larger timestamp has already been sent to the DM.
- If an operation is too late, the earlier operation cannot be undone, then the txn is aborted.
- The aborted txn is restarted with a new timestamp – why?
- This avoids cyclic restart.