Transactional Concurrency Control

Transactions: ACID Properties

“Full-blown” transactions guarantee four intertwined properties:

- **Atomicity**: Transactions can never “partly commit”; their updates are applied “all or nothing.”
  The system guarantees this using logging, shadowing, distributed commit.
- **Consistency**: Each transaction \( T \) transitions the dataset from one semantically consistent state to another.
  The application guarantees this by correctly marking transaction boundaries.
- **Independence/Isolation**: All updates by \( T_1 \) are either entirely visible to \( T_2 \), or are not visible at all.
  Guaranteed through locking or timestamp-based concurrency control.
- **Durability**: Updates made by \( T \) are “never” lost once \( T \) commits.
  The system guarantees this by writing updates to stable storage.

Isolation and Serializability

Isolation/Independence means that actions are *serializable.*
A schedule for a group of transactions is serializable iff its effect is the same as if they had executed in some serial order.

Obvious approach: execute them in a serial order (slow).

Transactions may be interleaved for concurrency, but this requirement constrains the allowable schedules:

- \( T_1 \) and \( T_2 \) may be arbitrarily interleaved only if there are no conflicts among their operations.
  - A transaction must not affect another that commits before it.
  - Intermediate effects of \( T \) are invisible to other transactions unless/until \( T \) commits, at which point they become visible.

Some Examples of Conflicts

A conflict exists when two transactions access the same item, and at least one of the accesses is a write.

1. **lost update problem**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): transfer $100 from B to C:
   \[ \text{R}(B) \quad \text{W}(B) \quad \text{R}(C) \quad \text{W}(C) \]

2. **inconsistent retrievals problem (dirty reads violate consistency)**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): compute total balance for A and C:
   \[ \text{R}(A) \quad \text{R}(C) \quad \text{W}(A) \]

3. **nonrepeatable reads**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): check balance and withdraw $100 from A:
   \[ \text{R}(A) \quad \text{R}(A) \quad \text{W}(A) \]

Serializable Schedules

A schedule is a partial ordering of operations for a set of transactions \( \{T, S, \ldots\} \), such that:

- The operations of each xaction execute serially.
- The schedule specifies an order for conflicting operations.
  Any two schedules for \( \{T, S, \ldots\} \) that order the conflicting operations in the same way are equivalent.

A schedule for \( \{T, S, \ldots\} \) is serializable if it is *equivalent* to some serial schedule on \( \{T, S, \ldots\} \).

There may be other serializable schedules on \( \{T, S, \ldots\} \) that do not meet this condition, but if we enforce this condition we are safe.

Conflict serializability: detect conflicting operations and enforce a serial-equivalent order.

Legal Interleaved Schedules: Examples

\( T \leq S \)

1. **avoid lost update problem**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): transfer $100 from B to C:
   \[ \text{R}(B) \quad \text{W}(B) \quad \text{R}(C) \quad \text{W}(C) \]

2. **avoid inconsistent retrievals problem**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): compute total balance for A and C:
   \[ \text{R}(A) \quad \text{R}(C) \quad \text{W}(A) \]

3. **avoid nonrepeatable reads**
   \( T \): transfer $100 from A to C:
   \[ \text{R}(A) \quad \text{W}(A) \quad \text{R}(C) \quad \text{W}(C) \]

   \( S \): check balance and withdraw $100 from A:
   \[ \text{R}(A) \quad \text{R}(A) \quad \text{W}(A) \]
Defining the Legal Schedules

1. To be serializable, the conflicting operations of $T$ and $S$ must be ordered as if either $T$ or $S$ had executed first.

   We only care about the conflicting operations: everything else will take care of itself.

2. Suppose $T$ and $S$ conflict over some shared item(s) $x$.

3. In a serial schedule, $T$’s operations on $x$ would appear before $S$’s, or vice versa...for every shared item $x$.

   As it turns out, this is true for all the operations, but again, we only care about the conflicting ones.

4. A legal (conflict-serializable) interleaved schedule of $T$ and $S$ must exhibit the same property.

   Either $T$ or $S$ “wins” in the race to $x$; serializability dictates that the “winner take all”.

The Graph Test for Serializability

To determine if a schedule is serializable, make a directed graph:

- Add a node for each committed transaction.
- Add an arc from $T$ to $S$ if any equivalent serial schedule must order $T$ before $S$.

   $T$ must commit before $S$ if the schedule orders some operation of $T$ before some operation of $S$.

   The schedule only defines such an order for conflicting operations...

   ...so this means that a pair of accesses from $T$ and $S$ conflict over some item $x$, and the schedule says $T$ “wins” the race to $x$.

   • The schedule is conflict-serializable if the graph has no cycles.

   (winner take all)

   $\begin{array}{c}
   T \\
   S \\
   A \\
   C \\
   S \\
   A \\
   T
   \end{array}$

The Graph Test: Example

Consider two transactions $T$ and $S$:

- $T$: transfer $100 from A to C: R(A) W(A) R(C) W(C)$
- $S$: compute total balance for A and C: $R(A)$ $R(C)$

$\begin{array}{c}
T: R(A) W(A) R(C) W(C) \\
S: R(A) R(C)
\end{array}$

$\begin{array}{c}
A \\
C
\end{array}$

(T total balance gains $100$)

$\begin{array}{c}
T: R(A) W(A) R(C) W(C) \\
S: R(A) R(C)
\end{array}$

$\begin{array}{c}
A \\
C
\end{array}$

(S total balance loses $100$)

Transactional Concurrency Control

Three ways to ensure a serial-equivalent order on conflicts:

- **Option 1**, execute transactions serially.
  - “single shot” transactions

- **Option 2**, pessimistic concurrency control: block $T$ until transactions with conflicting operations are done.
  - use locks for mutual exclusion
  - two-phase locking (2PL) required for strict isolation
  - Option 3, optimistic concurrency control: proceed as if no conflicts will occur, and recover if constraints are violated.
  - Repair the damage by rolling back (aborting) one of the conflicting transactions.

- **Option 4**, hybrid timestamp ordering using versions.

   $\begin{array}{c}
   T \\
   S \\
   A \\
   C \\
   S \\
   A \\
   T
   \end{array}$

Why 2PL?

If transactions are well-formed, then an arc from $T$ to $S$ in the schedule graph indicates that $T$ beat $S$ to some lock.

Neither could access the shared item $x$ without holding its lock.

Read the arc as “$T$ holds a resource needed by $S$”.

2PL guarantees that the “winning” transaction $T$ holds all its locks at some point during its execution.

Thus 2PL guarantees that $T$ “won the race” for all the locks...

...or else a deadlock would have resulted.

$\begin{array}{c}
T: R(A) W(A) R(C) W(C) \\
S: R(A) R(C)
\end{array}$

$\begin{array}{c}
A \\
C
\end{array}$
Why 2PL: Examples

Consider our two transactions $T$ and $S$:

$T$: transfer $100$ from A to C: 
\[ R(A) \quad W(A) \quad R(C) \quad W(C) \]

$S$: compute total balance for A and C: 
\[ R(A) \quad R(C) \]

Non-two-phased locking might not prevent the illegal schedules.

More on 2PL

1. 2PL is sufficient but not necessary for serializability.
   - Some conflict-serializable schedules are prevented by 2PL.
     $T$: transfer $100$ from A to C: 
     \[ R(A) \quad W(A) \quad R(C) \quad W(C) \]
     $S$: compute total balance for A and C: 
     \[ R(A) \quad R(C) \]

2. In most implementations, all locks are held until the transaction completes (strict 2PL).
   - Avoid cascading aborts needed to abort a transaction that has revealed its uncommitted updates.

3. Reader/writer locks allow higher degrees of concurrency.
4. Queries introduce new problems.
   - Want to lock “predicates” so query results don’t change.

Disadvantages of Locking

Pessimistic concurrency control has a number of key disadvantages, particularly in distributed systems:

- **Overhead.** Locks cost, and you pay even if no conflict occurs.
  - Even readonly actions must acquire locks.
  - High overhead forces careful choices about lock granularity.
- **Low concurrency.**
  - If locks are too coarse, they reduce concurrency unnecessarily.
  - Need for strict 2PL to avoid cascading aborts makes it even worse.
- **Low availability.**
  - A client cannot make progress if the server or lock holder is temporarily unreachable.
- **Deadlock.**

Optimistic Concurrency Control

OCC skips the locking and takes action only when a conflict actually occurs.
- Detect cycles in the schedule graph, and resolve them by aborting (restoring) one of the transactions.
- OCC always works for no-contention workloads.
  - All schedules are serial-equivalent.
- OCC may be faster for low-contention workloads.
  - We win by avoiding locking overhead and allowing higher concurrency, but we might lose by having to restart a few transactions. In the balance, we may win.
- OCC has drawbacks of its own:
  - Restarting transactions may hurt users, and an OCC system may thrash or “livelock” under high-contention workloads.

Inside OCC

There are two key questions for an OCC protocol:

- **Validation.** How should the system detect conflicts?
  - $T$ may not conflict with any $S$ that commits while $T$ is in progress.
  - Even readonly transactions must validate (no inconsistent retrievals).
- **Recovery.** How should the system respond to a conflict?
  - Restart all but one of the conflicting transactions.
  - Requires an ability to cheaply roll back updates.

Validation occurs when a transaction requests to commit:

- **Forward validation.** Check for conflicts with transactions that are still executing.
- **Backward validation.** Check for conflicts with previously validated transactions.

Serial Validation for OCC

OCC validation (forward or backward) is simple with a few assumptions:

- Transactions update a private “tentative” copy of data.
  - Updates from $T$ are not visible to $S$ until $T$ validates/commits.
- Transactions validate and commit serially at a central point.

Transaction manager keeps new state for each transaction $T$:
- Maintain a read set $R(T)$ and a write set $W(T)$ of items/objects read and written by $T$.

When $T$ first reads an item $x$, add $x$ to $R(T)$ and return the last committed value of $x$.
- $T$ cannot affect $S$ if $S$ commits before $T$, so we only need to worry about whether or not $T$ observed writes by $S$. 
**Serial Backward Validation for OCC**

When $T$ requests commit, check for conflicts with any previously validated transaction $S$.
- Examine transactions $S$ that committed since $T$ started.
  - Ignore transactions $S$ that commit before $T$ starts; $T$ will see all updates made by $S$ in this case.
- Verify that $R(T)$ does not conflict with any $W(S)$.
  - $T$ might not have observed all of the writes by $S$, as required for serializability.
- Write sets of candidate transactions $S$ are kept on a validation queue. (for how long?)
- If $T$ fails validation, restart $T$.

**Backward Validation: Examples**

Consider two transactions $S$ and $T$:

- $S$: transfer $\$100$ from $A$ to $C$.
- $T$: compute total balance for $A$ and $C$: $R(A)$ $W(A)$ $R(C)$ $W(C)$

Note: $S$ cannot affect $T$ if $T$ commits first, since writes are tentative.

**Distributed OCC**

The scenario: client transactions touch multiple servers.

- Thor object server (Adya, Gruber, Liskov, Maheshwari, SIGMOD'95)
- Problem 1 (client/server): If the client caches data and updates locally, the cache must be consistent at start of each transaction.
  - Otherwise, there is no guarantee that $T$ observes writes by an $S$ that committed before $T$ started.
- Validation queue may grow without bound.
- Problem 2 (multiple servers): validation/commit is no longer serial.
  - There is no central point at which to order transactions by their entry into the validation phase.
  - How to ensure that transactions validate in the same order at all sites, as required for serializability?

**Client/Server OCC in Thor**

**Key idea:** use cache callbacks to simplify validation checks, while allowing clients to cache objects across transactions.

- Each server keeps a conservative cached set of objects cached by each client $C$.
- If a validated $S$ has modified an object $x$ in $C$’s cached set:
  1. Callback client to invalidate its cached copy of $x$.
  2. Client aborts/restarts any local action $T$ with $x$ in $R(T)$.
  3. Add $x$ to $C$’s invalid set until $C$ acks the invalidate.
- A transaction $T$ from client $C$ fails validation if there is any object $x$ in $R(T)$ that is also in $C$’s invalid set.

**Multiple-Server OCC in Thor**

Transactions validate serially at each server site.

- The server validates the transactions with respect to the objects stored on that server, during the voting phase of 2PC.
- Validations involving multiple servers may be overlapped.
  - Must validate the same conflict-serial order at all servers.
- **Key idea:** timestamp transactions as they enter validation at a client coordinator, and order/validate them in source timestamp order.
  - Any site may act as coordinator for any transaction.
  - Timestamps are based on loosely synchronized physical clocks.
  - The protocol is simple if transactions arrive for validation in timestamp order at each site, but they might not...

**In-Order Validation**

If transactions arrive in timestamp order, standard conflict checks are sufficient:

To validate $T$ against $S < T$, check standard conflict condition:

- $W(S)$ intersects $R(T) = \emptyset$

- If $S$ committed before $T$ arrives from client $C$, checking the invalid set is just as good as the standard conflict check:
  - $R(T)$ intersects invalid set of $C$ = $\emptyset$
- If $S$ has validated but is not locally known to have committed:
  - $S$ is in the prepared state; its $W(S)$ has not yet been merged into $C$’s invalid set.
  - We still need that validation queue...
Out-of-Order Validation

Suppose $S$ arrives for validation at some site and finds a validated $T$ present with $S < T$.
- $S$ arrived “late”: it must be ordered before $T$.
  If it’s too late to rely only on in-order checks at this site, even though other sites may have used them to validate $S < T$.
- If $T$ has validated, this site can no longer abort $T$ unilaterally.
  Use standard check, but abort $S$ instead of $T$ if there’s a conflict.
  Now we must also keep read sets on the validation queue.
- What if $T$ has already committed? (we might not even know)
  Is the standard check sufficient?

The “Problem” of External Consistency

Suppose again that $S$ encounters a validated $T$ with $S < T$.
- Oops: clock skew...
  We know we need the standard check, but...
  - $T$ might have committed before $S$ even started!
    We must order $T$ before $S$ even if the timestamps say differently, in order to preserve external consistency.
    External consistency: “What you see is what you get.”
  - Claim: must check for conflicts in both directions in this case.
    Fail $S$ if its read set conflicts with $T$’s write set.
    i.e., Fail $S$ if it has any conflict with a validated $T$ and $S < T$.
  - Managing that validation queue just got even harder....

VQ Truncation in Thor’s OCC

We cannot keep transactions on the validation queue forever.
- We need only prepared transactions for the in-order checks.
  Invalid sets reflect updates from committed transactions.
- We need committed transactions on the VQ only for the out-of-order checks.
  How long should we hold the door open for late arrivals?

Solution: discard old VQ records and conservatively abort transactions that arrive too late for the needed checks.
- Keep $\text{threshold} = \text{[timestamp of the latest discarded } T\text{]}$.
- If $S$ arrives with $S < \text{threshold}$, abort $S$.
  ($S$ must be checked against transactions long forgotten.)

Review: Multi-Server OCC in Thor

Transactions validate serially at each server site, and validations involving multiple servers may be overlapped.
Most validate in the same order at all sites, and must preserve external consistency if skewed clocks cause the timestamps to lead us astray.

Solution: timestamp transactions as they enter validation at their coordinator, and try to validate/commit in timestamp order.
Timestamps are based on loosely synchronized physical clocks.
The protocol is simple if transactions arrive for validation in timestamp order at each site, but they might not...
If clocks are skewed, some transactions may restart unnecessarily.
We can tolerate higher clock skew, but only at a cost.