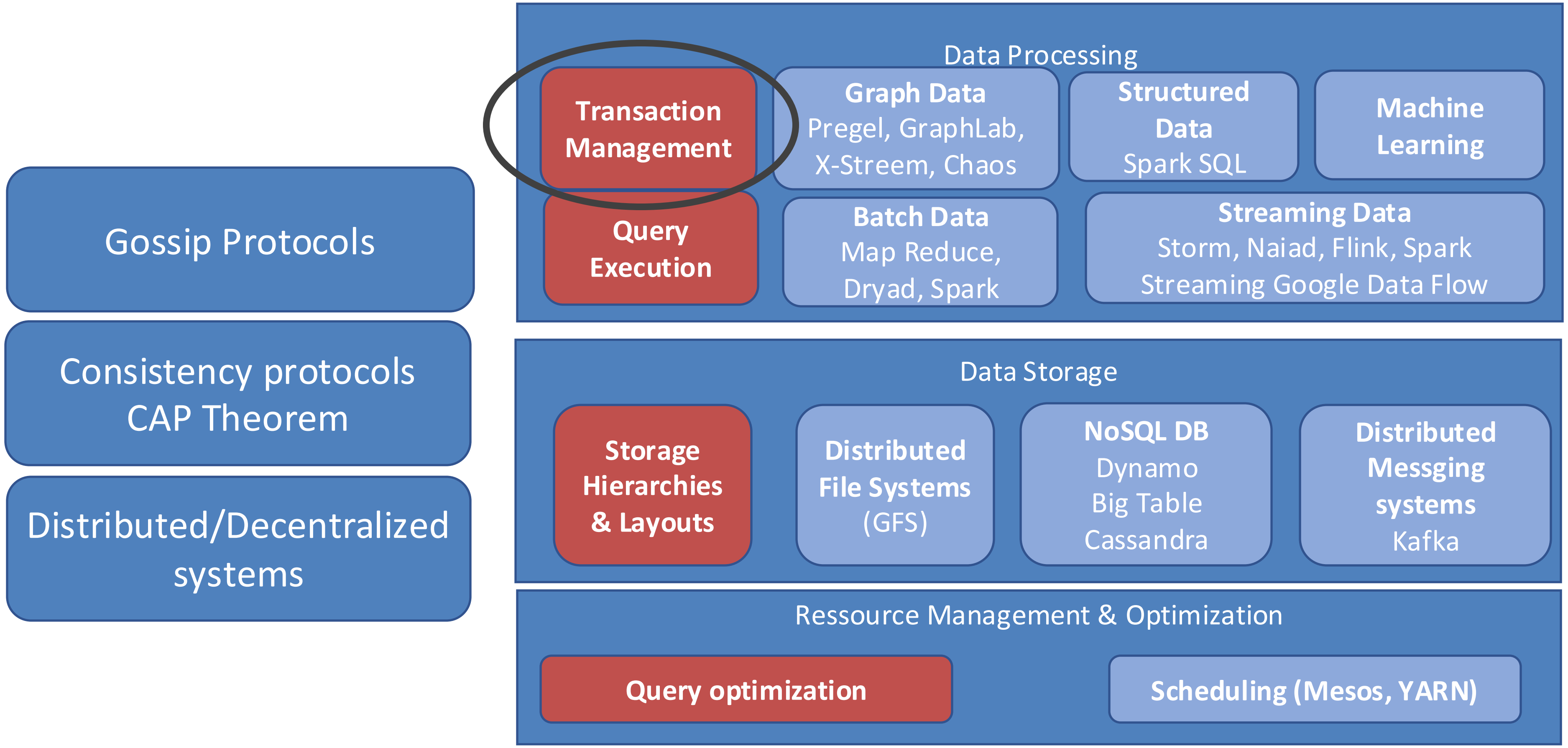
CS460
Systems for Data Management and Data Science

Concurrency control

Prof. Anastasia Ailamaki
Data-Intensive Applications and Systems (DIAS) Lab

Today's topic

Data science software stack



Outline

- Transactions & Concurrency Control
  - **ACID & Transaction Schedules**
  - Concurrency control protocols
    - Pessimistic
    - Optimistic
  - Multi-version concurrency control

Definition of transactions

A transaction (txn, or Xact) is a sequence of actions that are executed on a shared database to perform some higher-level function.

Transactions are the basic unit of change in the DBMS. No partial txns are allowed.

A quick reminder of ACID

- Atomicity: Either all actions in the txn happen, or none.
- Consistency: If each txn is consistent, and the DB starts consistent, it ends up consistent.
- Isolation: Execution of one txn is isolated from other txns.
- Durability: If a txn commits, its effects persist.

Atomicity and Durability

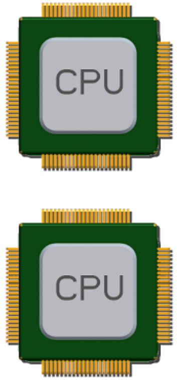
- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- All transactions are *atomic*.
  - A user can think of a txn as always executing all its actions in one step, or not executing any actions at all.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- Durability also relies on logs

Consistency and Isolation

- Each transaction must leave the database in a consistent state.
  - DBMS will enforce some integrity constraints.
  - Clearly, no semantic consistency.
- Users submit transactions, and expect isolation -- each transaction executed by itself.
  - **Concurrency** very important for performance: interleaving actions from different transactions.
  - Net effect identical to executing all transactions one after the other in some serial order.

A note on concurrency

- Several transactions arrive at (almost) the same time
- Need to execute in parallel to fully utilize hardware



T1: R(A) R(B) compute-something W(C) COMMIT

T2: R(E) R(A) R(D) compute-something …

T3: R(D) compute-something R(E) …

User writes SQL queries.
Translated to actions!

Schedules

- The DBMS gets as input a set of transactions and executes a schedule.

- Schedule: a list of actions (reading, writing, aborting, or committing) from a set of txns
  - All actions appear in the schedule
  - The order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T.

Example

T1: transfer $100 from B's account to A's account.

T2: credit both accounts with a 6% interest payment.

```
T1:    BEGIN   A=A+100,    B=B-100     END
T2:    BEGIN   A=1.06*A,   B=1.06*B    END
```

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- Actions of two transactions can interleave!
- However, the net effect **must be equivalent** to these two transactions running serially in some order.

Example (Contd.)

- Consider a possible interleaving schedule:

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- This is OK. But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- The system's view of the second schedule:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Time →

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

If each transaction preserves consistency, every serializable schedule preserves consistency.

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

Time →

Aborting a Transaction

- If T_i is aborted, all its actions have to be undone.
- Cascading aborts: If T_j reads an object last written by T_i, T_j must be aborted as well!
  - Alternative to avoid cascading aborts: If T_i writes an object, T_j can read this only after T_i commits.
- DBMS maintains a write log, in order to be able to *undo* the actions of aborted txns.
  - Also used to recover from system crashes: all active txns at the time of the crash are aborted when the system comes back up.

Conflict Serializable Schedules

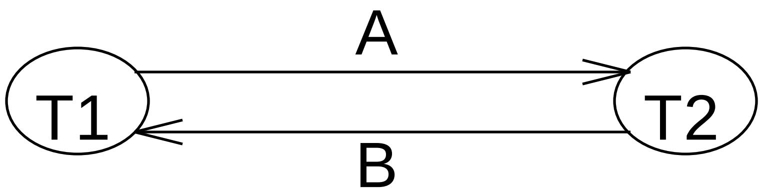
- Two schedules are conflict equivalent if:
  - They involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
  - i.e., we can transform one into the other by swapping non-conflicting adjacent operations

- Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule.
  - Intuition: You can transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	



Dependency graph
a.k.a. precedence graph

- *Precedence graph*: One node per txn; edge from T_i to T_j if T_j reads/writes an object last read/written by T_i.
- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Precedence Graph

- Also known as dependency graph/ serializability graph

- *Precedence graph*: One node per txn; edge from T_i to T_j if T_j reads/writes an object last read/written by T_i.

- Theorem: A schedule is conflict serializable if and only if its dependency graph is acyclic

Outline

- Transactions & Concurrency Control
  - ACID & Transaction Schedules
  - **Concurrency control protocols**
    - **Pessimistic**
    - Optimistic
  - Multi-version concurrency control

Concurrency protocols

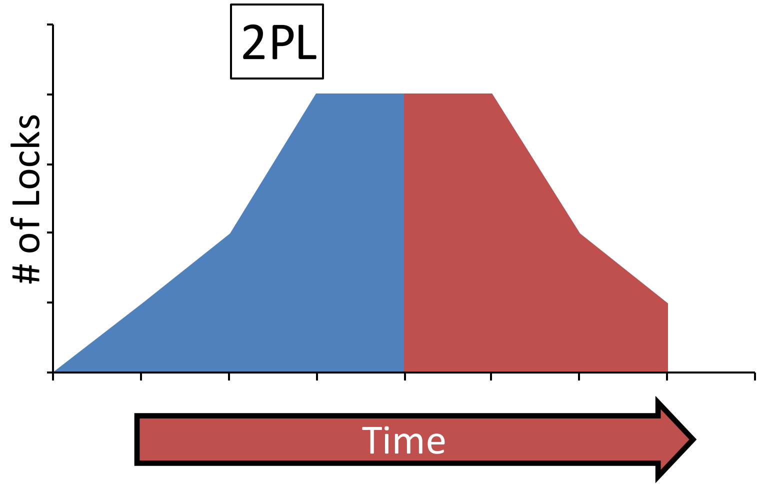
- Two-phase locking (2PL)
  - Pessimistic approach
  - Assume txns will conflict!
  - Acquire locks on items before accessing them!

Lock-Based Concurrency Control

- Transactions acquire locks before reading and writing
- Locking protocol guarantees that schedule will be conflict serializable (*correct)* if it completes
  - Deadlocks are possible
- Locking granularity can be anything
  - Tables, indexes, pages, records

Lock-Based Concurrency Control

Two-Phase Locking (2PL) Protocol



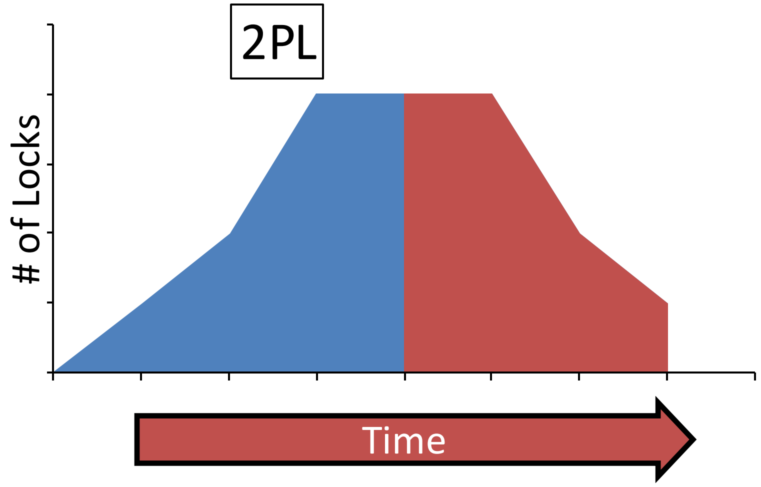
- Rule 1: Each txn obtains
  - S (shared) lock before reading
  - X (exclusive) lock before writing
  - Sometimes also called read/write locks
- Rule 2: A txn cannot request additional locks once it releases any locks.
- 2PL allows only schedules whose precedence graph is acyclic => serializable.

Example schedule with locks:

T1: S(A) R(A) S(B) R(B)

T2: X(C) R(C) W(C) S(D) R(D)

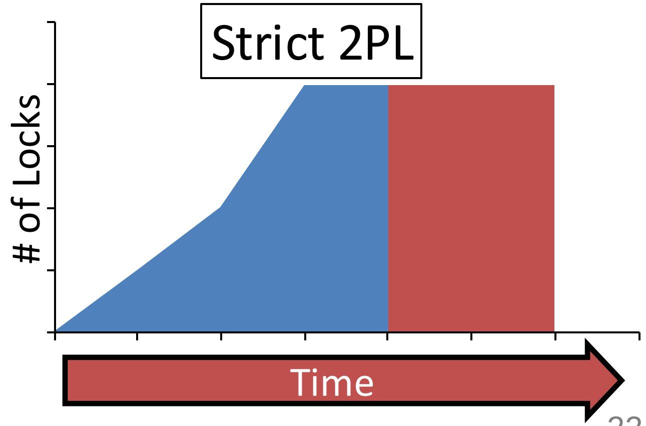
Lock-Based Concurrency Control



Two-Phase Locking (2PL) Protocol

- Rule 1: Each txn obtains
  - S (shared) lock before reading
  - X (exclusive) lock before writing
  - Sometimes also called read/write locks
- Rule 2: A txn cannot request additional locks once it releases any locks.
- 2PL allows only schedules whose precedence graph is acyclic => serializable.

***Strict** Two-phase Locking (Strict 2PL) Protocol*



- Rule 3: All locks released when the txn completes.
- Strict 2PL additionally simplifies transaction aborts
  - (Non-strict) 2PL involves more complex abort processing.

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.

- Two ways of dealing with deadlocks
  - Deadlock detection: detect and resolve deadlocks when they are created.
  - Deadlock prevention: never let deadlocks happen.

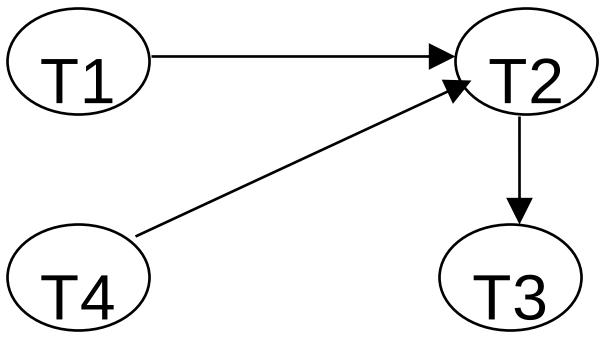
Deadlock Detection

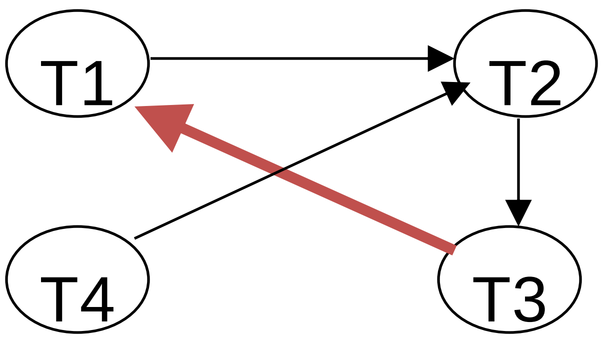
- If a lock request cannot be satisfied, the transaction is suspended and must wait until the resource becomes available.
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection (Continued)

Example:

T1:	S(A)	R(A)			S(B)					
T2:			X(B)	W(B)				X(C)		
T3:						S(C)	R(C)			X(A)
T4:									X(B)	





Deadlock Prevention

- Assign priorities based on timestamps.
  - Earlier timestamp → higher priority
- Assume T_i wants a lock that T_j holds.
  Two policies:
  - Wait-Die ("old waits for young"):
    - If T_i has higher priority, T_i waits for T_j. Otherwise T_i aborts
  - Wound-wait ("young waits for old"):
    - If T_i has higher priority, T_j aborts. Otherwise T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Fixed vs dynamic databases

- Fixed tuples
  - Set of tuples is fixed
  - Can update, but no inserts/deletes
  - Can lock all related tuples

```
UPDATE employees
        SET salary=salary*1.2 WHERE age>60
```

- Dynamic databases
  - Can insert/delete tuples
  - Cannot lock all related tuples
  - `INSERT INTO employees(name,age,salary) VALUES("Superman", 62, 10000)`

Dynamic Databases

- If insertions/deletions are allowed, then even Strict 2PL **cannot** assure serializability
  - T1: Print the oldest sailors with rating=1 and rating=2
  - T2: Insert a sailor with (rating=1,age=96), and delete the oldest sailor with rating=2
  - The results **may not** correspond to a serial execution → not conflict-serializable!

Dynamic Databases

- If insertions/deletions are allowed (not only updates), then even Strict 2PL **cannot** assure serializability
  - T1 **locks all pages** containing sailor records with rating = 1, and finds oldest sailor (age = 71).
  - Next, T2 **inserts a new sailor**; rating = 1, age = 96.
  - T2 also **deletes** oldest sailor with rating = 2 (age = 80), and commits.
  - T1 now locks all pages containing sailor records with rating = 2, and finds oldest (age = 63).
- Not conflict serializable!!!

T1:	S(A*) R(A*)		S(B*) R(B*) W(C)
T2:		X(A') I(A') X(B) D(B)	

How did serializability break aka "The phantom menace"

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!

- Conflict serializability guarantees serializability only if the set of objects is fixed!
- Need some mechanism to enforce this assumption.
  - (Index locking and predicate locking).

Predicate Locking

- Implicitly lock all records (also new) that satisfy a logical predicate
  - I.e., rating=1 or rating=2.

- How would you implement predicate locking?
  - Very expensive

Index Locking

- T1 locks the index pages containing the data entries with *rating* = 1 and *rating* = 2.
  - Index needs to be updated after the insert → will fail if locked
  - If there are no records with rating = 1, T1 must lock the index page where such a data entry would be, if it existed!

- Special case of predicate locking – more efficient implementation.

Are 2PL protocols always good?

- Locking: Conservative approach in which conflicts are prevented.
- Disadvantages
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, gain concurrency by not locking, and instead checking for conflicts before txns commit

Outline

- Transactions & Concurrency Control
  - ACID & Transaction Schedules
  - **Concurrency control protocols**
    - Pessimistic
    - **Optimistic**
  - Multi-version concurrency control

Concurrency protocols

- Two-phase locking (2PL)
  - Pessimistic approach
  - Assume txns will conflict!
  - Acquire locks on all items before accessing them!

- Timestamp ordering (T/O)
  - Optimistic approach
  - Assume that conflicts are rare!
  - Do not acquire locks, check for conflicts at commit!

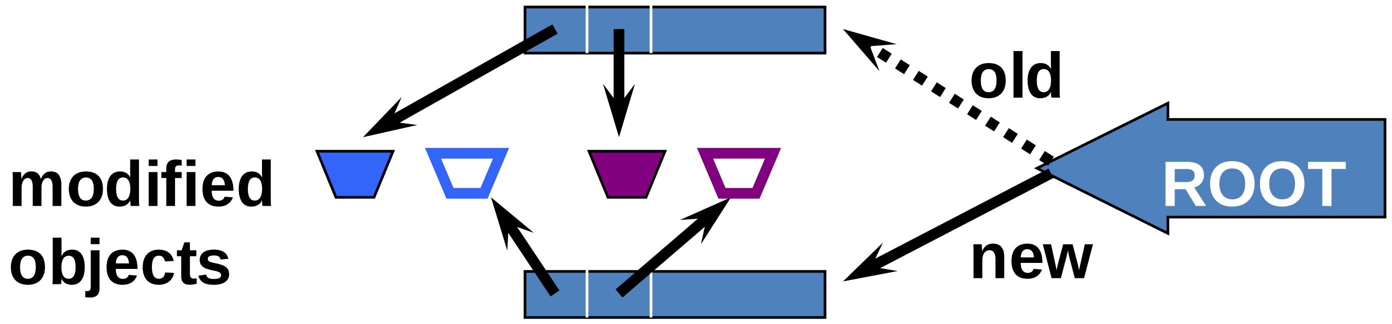
Optimistic Concurrency Control

The Kung-Robinson Model

- Key idea: Timestamp ordering is imposed on transactions, and validation phase checks that all conflicting actions occurred in the same order.
- If this is not the case, abort the transaction that started later!

Kung-Robinson Model

- Txns have three phases:
  - READ: txns read from the database, but make changes to **private copies** of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

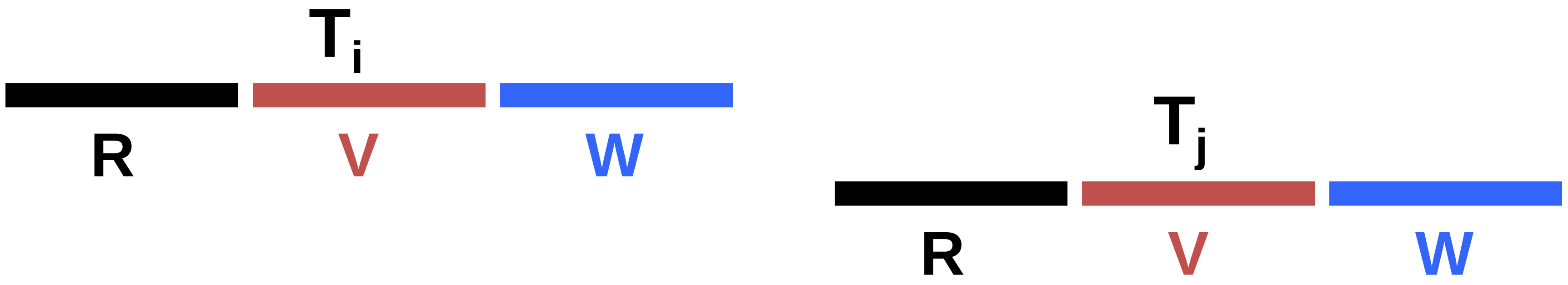


Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each txn assigned a numeric id.
  - Just use a **timestamp**.
  - Txn ids assigned at end of READ phase, just before validation begins.
- ReadSet(T_i): Set of objects read by txn T_i.
- WriteSet(T_i): Set of objects modified by T_i.

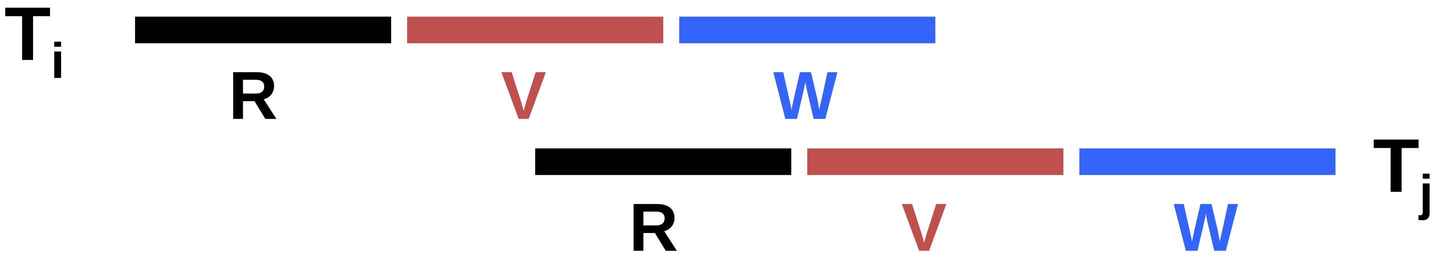
Test 1

- For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins.



Test 2

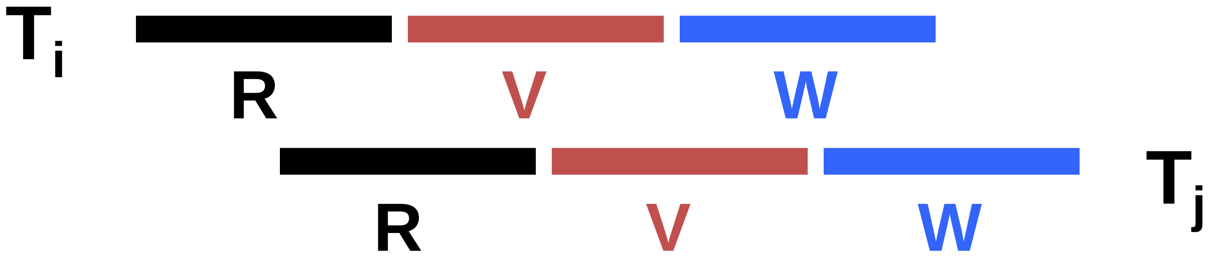
- For all i and j such that $T_i < T_j$, check that:
  - T_i completes before T_j begins its Write phase
  - $WriteSet(T_i) \cap ReadSet(T_j)$ is empty.



✓ **“Does T_j read dirty data?”**

Test 3

- For all i and j such that $T_i < T_j$, check that:
  - T_i completes Read phase before T_j does **+**
  - WriteSet(T_i) ∩ ReadSet(T_j) is empty **+**
  - WriteSet(T_i) ∩ WriteSet(T_j) is empty.



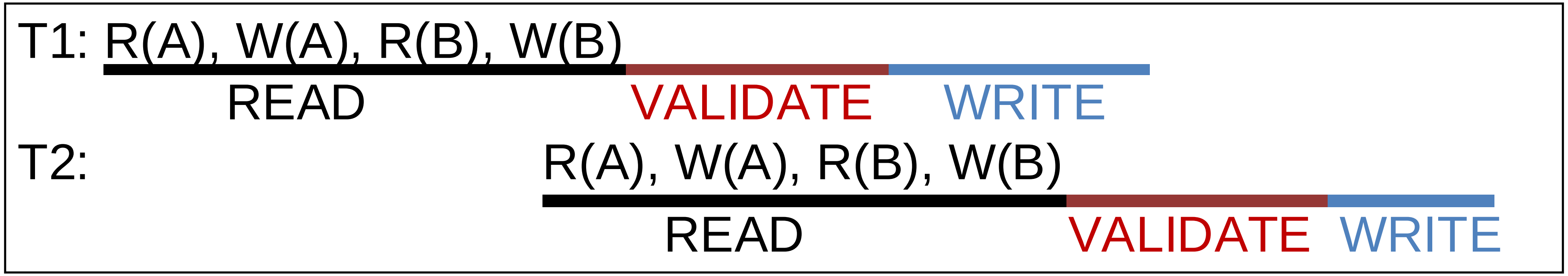
✓ "Does T_j read dirty data?"
✓ "Does T_i overwrite T_j's writes?"

Example: Optimistic CC

T1: R(A), W(A), R(B), W(B) C
T2: R(A), W(A), R(B), W(B) C

T1: R(A), W(A), R(B), W(B)
READ VALIDATE WRITE
T2: R(A), W(A), R(B), W(B)
READ VALIDATE WRITE

Example: Optimistic CC



- Validation of T2:
  - Test 1: ???

- For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins.

Example: Optimistic CC

T1: R(A), W(A), R(B), W(B)
READ VALIDATE WRITE

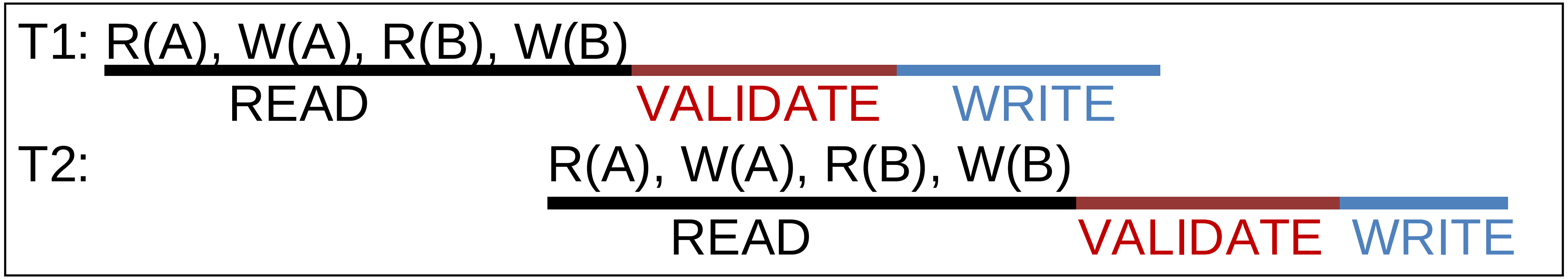
T2: R(A), W(A), R(B), W(B)
READ VALIDATE WRITE

- Validation of T2:
  - Test 1: fails   Test 2: ???

For all i and j such that $T_i < T_j$, check that:

- T_i completes before T_j begins its Write phase
- WriteSet(T_i) ⋂ ReadSet(T_j) is empty.

Example: Optimistic CC

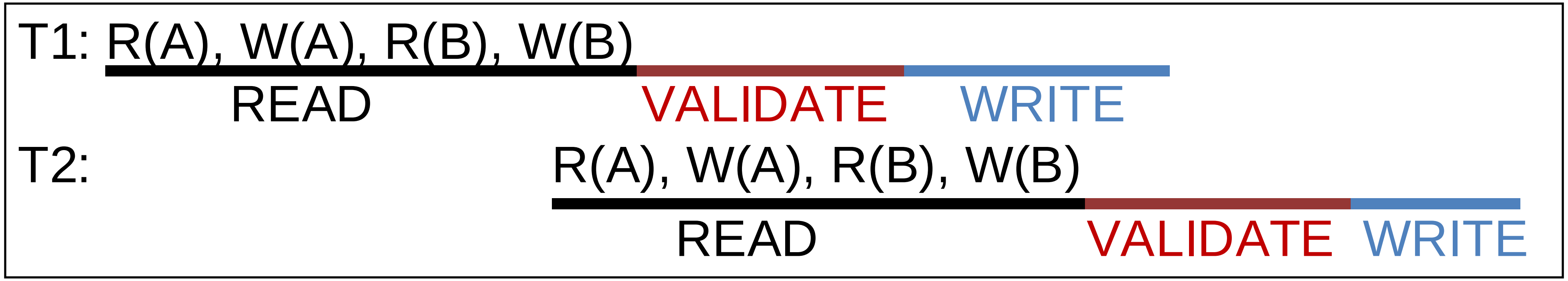


- Validation of T2:
  - Test 1: fails   Test 2: fails   Test 3: ???

For all i and j such that $T_i < T_j$, check that:

- T_i completes Read phase before T_j does **+**
- WriteSet(T_i) ∩ ReadSet(T_j) is empty **+**
- WriteSet(T_i) ∩ WriteSet(T_j) is empty.

Example: Optimistic CC



- Validation of T2:
  - Test 1: fails Test 2: fails Test 3: fails

T2 gets restarted once T1 is completely finished

Comments on Validation

- Assignment of txn id, validation, and the Write phase are inside a **critical section**!
  - Nothing else goes on concurrently.
  - If validation/write phase is long, major drawback!

- Optimization for Read-only txns:
  - Shorter critical section
    (because there is no Write phase).

Overheads in Optimistic CC

- Record read/write activity in ReadSet and WriteSet per txn.
  - Must create and destroy these sets as needed.
- Check for conflicts during validation, and make validated writes “global”.
  - Critical section can reduce concurrency.
- **Optimistic CC restarts txns that fail validation**.
  - Work done so far is wasted.
  - Requires clean-up.

Timestamp-based CC

- Optimistic CC: Timestamp ordering is imposed on transactions, and **validation** checks that all conflicting actions occurred in the same order.
- Timestamp-based CC
  - **Continuous validation – not a distinct phase**
  - Keep read and write timestamps **per object**, and starting timestamp of each txn
  - Compare txn timestamp with read/write timestamps of the objects in order to decide between:
    - Continue, Abort, Commit, Skip write

Continuous, per-object validation

Timestamp-based CC

Idea:

- Txn timestamp TS ← begin time
- Object: read-timestamp (RTS) and a write-timestamp (WTS)
  - If action a_i of txn T_i conflicts with action a_j of txn T_j, and $TS(T_i) < TS(T_j)$, then a_i must occur before a_j. Otherwise, restart violating txn.
  - Use RTS, WTS to detect conflicts.

Object	Read-timestamp	Write-Timestamp
A	10	4
B	15	13
…	…	…

When txn T wants to READ Object O

- TS(T) < WTS(O): violates timestamp order of T w.r.t. writer of O.
  - Abort T and restart it with a new, higher TS.
- TS(T) >= WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written in some persistent fashion → overhead.

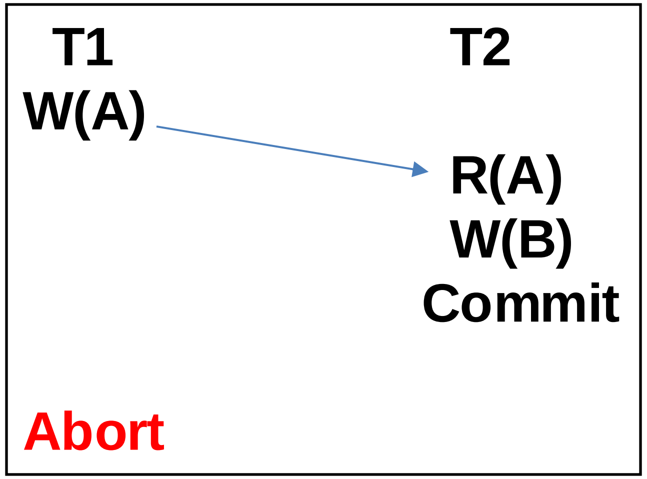
When txn T wants to write Object O

- TS(T) < RTS(O): violates timestamp order of T w.r.t. reader of O → abort and restart T.
- TS(T) < WTS(O) → violates timestamp order of T w.r.t. writer of O. → ???
  - Thomas Write Rule: Outdated write → Safely ignore the write –it's as if the write happened before and was overwritten
  - need not restart T!
  - Allows some **serializable schedules (correct) that are not conflict serializable**.

- Else, allow T to write O (and update WTS(O)).

Timestamp-based CC and Recoverability

- Unrecoverable schedules are possible



- Solution
  - Make changes of T1 in a memory buffer and block T2 from committing
  - Write changes to disk ONLY at commit

Outline

- Transactions & Concurrency Control
  - ACID & Transaction Schedules
  - Concurrency control protocols
    - Pessimistic
    - Optimistic
  - **Multi-version concurrency control**

Multiversion Concurrency Control

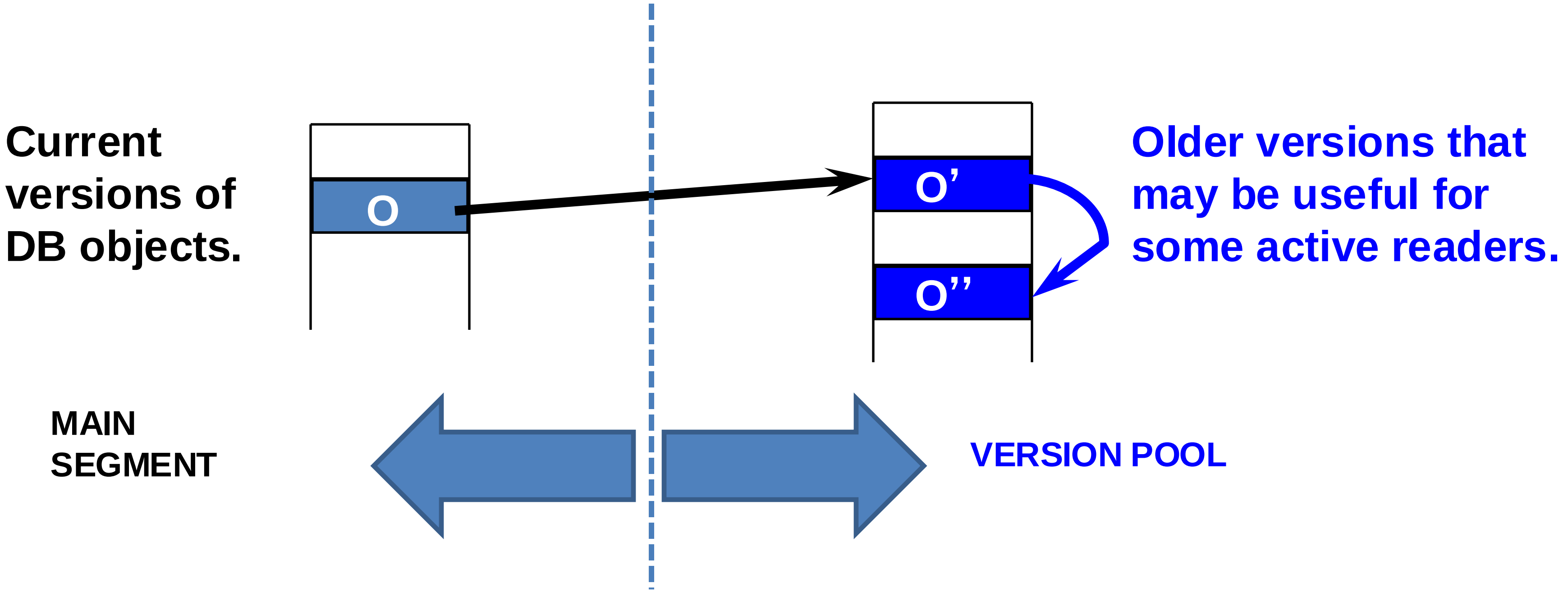
- **Goal: A transaction never waits on read!**
- **Idea**
  - Maintain **several versions** of each database object (multi-version), each with a read and a write timestamp.
  - Transaction T_i reads the most recent version whose write timestamp precedes $TS(T_i)$,
    i.e., $WTS(O) < TS(T_i)$.

Multiversioning is a storage mgmt concept!

Combine with CC => MVTO, MVOCC, MV2PL

Multiversion Concurrency Control

MVCC lets writers make a “new” copy while readers use an appropriate “old” copy:



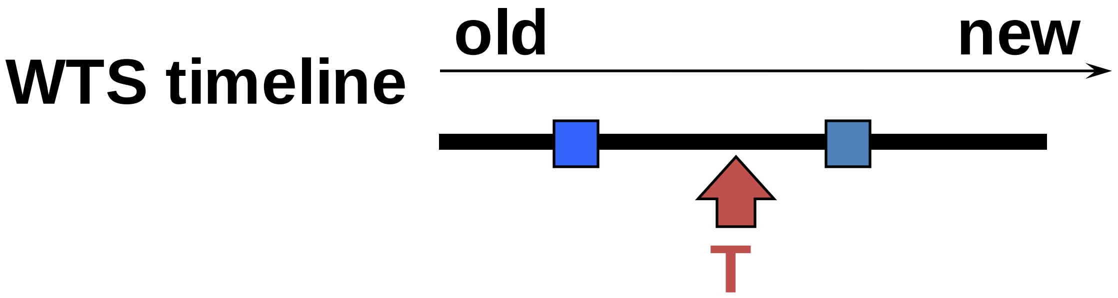
MV + Timestamp Ordering (MVTO)

For each version

- WTS: the timestamp of the txn that created it
- RTS: the timestamp of the txn that last read it
- Versions are (usually) chained backward; we can discard versions that are "too old to be of interest" (i.e., garbage collection).
- Each txn is classified as Reader or Writer.
  - Writer may write some object; Reader never will.
  - Txn declares whether it is a Reader when it begins.
- Readers are always allowed to proceed.

Reader txn

- For each object to be read:
  - Finds **newest version** with WTS < TS(T): Starts with current version in the main segment and chains backward through earlier versions.
  - Updates RTS to MAX(RTS, TS(T)).
- Reader txns are never restarted.



Readers always proceed

Writer txn

- To read an object, follow reader protocol.
- To write an object:
  - Finds **newest version V**
  - RTS(V) > TS(T): Reject write
  - RTS(V) ≤ TS(T): T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T).
    - Write is buffered / locked until T commits; other txns cannot read version CV. WHY????

Writers create new copy

(…if no younger transaction has read the data)

(…and if no active xaction holds V's lock)

Bottlenecks

- Lock thrashing
  - 2PL, Strict 2PL
- Timestamp allocation
  - All T/O algorithms + deadlock prevention
- Memory allocation
  - MVCC, OCC

Improving performance of txn

Goal is to

- reduce conflicts
- reduce time spent on each transaction

Three key approaches

- Stored procedures --> faster
- Prepared statements --> precompiled
- Query batches --> batch locking

Snapshot isolation

- Snapshot isolation (SI) is the most popular **isolation guarantee** in real DBMS.
  - all txn reads will see a consistent snapshot of the database
  - the txn successfully commits only if no updates it has made conflict with any concurrent updates made since that snapshot.
- SI does not guarantee serializability!
  - SerializableSI: Stronger, more conservative protocol
- Implemented in Oracle, MS SQL Server, Postgres.

Snapshot isolation

- Conceptually, txn works on a copy of the db made at txn start time.
  - Very expensive → not implemented that way but still expensive.
  - Guarantees that reads in the txn see a consistent version of the db.
- At commit time, verify that the values changed by the transaction have not been changed by other transactions since the snapshot was taken.
- *Write skew* anomaly
  - Not serializable, but permitted by snapshot isolation!

T1:	R(X)R(Y)		W(X) C	
T2:		R(X)R(Y)		W(Y) C

Write skew – (more concrete) example

[Source: Martin Kleppmann]

Alice:

begin transaction

```
currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
```

Now currently_on_call = 2

```
if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Alice'
  and shift_id = 1234
}
```

commit transaction

name	on_call
Alice	true
Bob	true
Carol	false

Alice	false

Bob	false

name	on_call
Alice	false
Bob	false
Carol	false

Bob:

begin transaction

```
currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
```

Now currently_on_call = 2

```
if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Bob'
  and shift_id = 1234
}
```

commit transaction

Discussion

- SI is related to optimistic CC, in that
  - Conceptually, snapshots are created at txn start.
  - There is an analysis phase at the end to decide whether a transaction may commit (do writesets overlap?).

- Multiverson CC is a way to implement (a stronger) snapshot isolation.