CC Protocols

Locking Policies

- Several locking policies consider efficiency and fairness
- writer starvation problem, deadlock problem

Writer Starvation Problem

- If several READ requests are compatible, immediately grant the lock request. Such policy may cause writer starvation problem if there are a large number of read requests. If a reader was granted a read lock, then fellow readers can immediately join in. However, writers will be blocked out, until all readers have finished. In fact, some unlucky writers may get blocked indefinitely.
- A solution is to use FIFO (first in, first out) policy to queue up the requests. Only requests at the front of the queue can try to get the lock. However, concurrency and efficiency may be negatively impacted. Some opportunities for parallel access will be lost as queue processing is serial in nature.

Deadlock Problem

- lock granting priority must be given to the parties who already own some kind of locks
- Time-out followed by rollback of the transaction would cause the release of the locks responsible for the deadlock

Optimistic CC

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.

Validation-Based Protocol

- Execution of transaction Ti is done in three phases.

  Read and execution phase Transaction Ti writes only to temporary local variables

  Validation phase

  Transaction Ti performs a ``validation test'' When transaction wants to commit, DBMS checks whether transaction would possibly have conflicted with any other concurrently running transactions. If there is a possible conflict, transaction is aborted,

  Private wokspace is cleared and it is restarted

  Write phase In case of no conflict, changes to the data item made in private workspace are copied to the database

Validation-based protocol

- Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation
- In case of few conflicts, validation can be done efficiently and leads to better performance than locking
- If there are many conflicts, cost of repeatedly restarting transactions hurts performance
- No checking is done while transaction is executing
- Updates by the transaction are not directly applied to the database items until transaction reaches end
- intermediate changes are made to the local copies and at validation it is checked for possible conflicts (serializability violation)

Kung-Robinson Model

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

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps:
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each transaction is assigned a numeric id.
  - use a **timestamp**.
- transaction ids assigned at the beginning of validation phase
- ReadSet(Ti): Set of objects read by transaction Ti.
- WriteSet(Ti): Set of objects modified by Ti.

Validation contd…

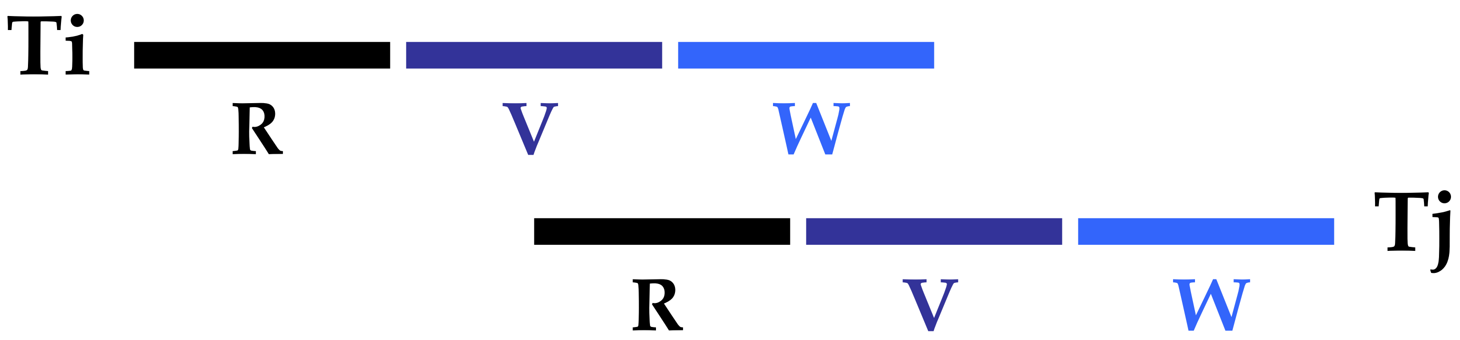
- Validation criterion checks whether the timestamp ordering of transactions is an equivalent serial order
- To validate Tj, one of the validation conditions must hold with respect to each committed transaction Ti such that TS(Ti) < TS(Tj)
- Validation conditions :
  - Each condition ensures that Tj’s modifications are not visible to Ti
  - To validate Tj we must check to see that one of the validation conditions holds with respect to each committed transaction Ti such that TS(Ti) < TS(Tj)
  - At most one transaction is in validation/write phases at any time

Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Test 2

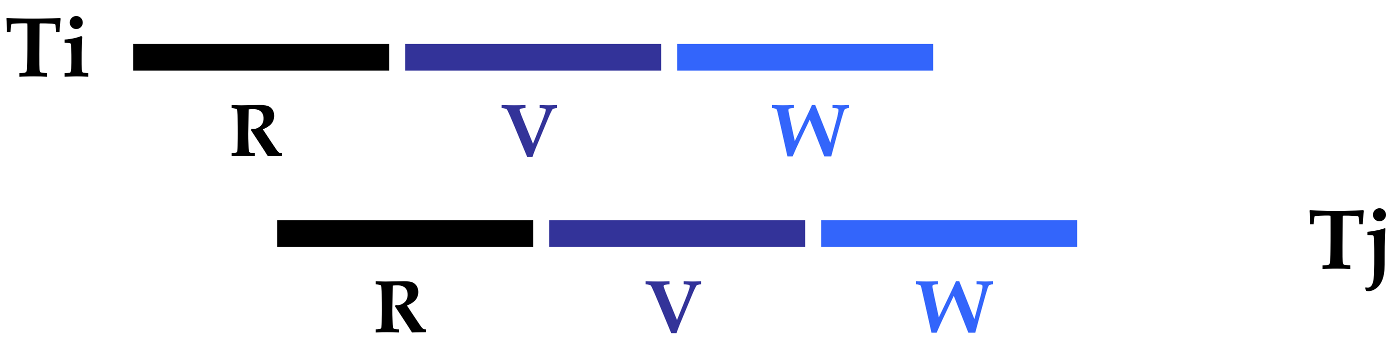
- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase **+**
  - WriteSet(Ti) **intersection** ReadSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does **+**
  - WriteSet(Ti) **Intersection** ReadSet(Tj) is empty **+**
  - WriteSet(Ti) **Intersection** WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per transaction.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
- Optimistic CC restarts transactions that fail validation.
  - Work done so far is wasted; requires clean-up.

Optimistic 2PL

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of transaction, make writes global, then release all locks.
- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in transactions being blocked, waiting for locks.
  - However, no validation phase, no restarts

Timestamp CC

- In lock based CC , conflicting actions of different transactions are ordered by the order in which locks are obtained, this is extended to actions (using lock protocols) and hence serializability is achieved
- In optimistic CC timestamp ordering checks are done for the stamps for conflicting actions of transactions
- **Timestamp based CC :** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each transaction a timestamp (TS) when it begins:
- If action ai of transaction Ti conflicts with action aj of transaction Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating transaction

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Contd..

- Every database object is given:
- Read timestamp RTS(O)
- Write timestamp WTS (O)

When transaction T wants to read Object O

- If TS(T) < WTS(O), the order of this read with respect to the most recent write on O would violate the TS order between this transaction and the writer
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again! Contrast use of timestamps in 2PL for ddlk prevention)
- If 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 to disk and recorded in the log.
- Log entry and restarts represent overheads.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q):
  1. If $TS(T_i) \leq$ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If $TS(T_i) \geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), $TS(T_i)$).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write** (Q).
  1. If $TS(T_i)$ < R-timestamp (Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If $TS(T_i)$ < W-timestamp (Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to $TS(T_i)$.

When transaction T wants to Write Object O

- If TS(T) < RTS(O), the write action conflicts with the most recent read action of O, and T is aborted and restarted.
- If TS(T) < WTS(O), the write of T conflicts with the most recent write of O and is out of timestamp order
- Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T's write is effectively followed by another write, with no intervening reads.) Allows some serializable but nonconflict serializable schedules:
- Else, allow T to write O and WTS(O) is set to TS(T)

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

Thomas Write Rule

- Ignoring outdated writes
- it states that, if a more recent transaction has already written the value of an object, then a less recent transaction does not need perform its own write since it will eventually be overwritten by the more recent one.
- If $TS(T) < WTS(O)$, the current write action has been made obsolete by the most recent write of O, which follows the current write according to the timestamp ordering
- It is as if T's write action had occurred immediately before the most recent write of O and hence was never read by anyone
- If TRL is not used and T is aborted (when $TS(T)<WTS(O)$, the protocol like 2PL will allow only conflict serializable schedules
- Use of TRL will allow some schedules which are not conflict serializable

TRL (2)

- Serializable schedule
- Not conflict serializable
- T2's write follows T1's read and precedes T1's write of the
  same object (non conflict serializable because writes of T1 and T2 ordering is different)

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

TRL (3)

- TRL relies on the observation that T2's write is never seen by any transaction and therefore the write action of T2 can be deleted to make the schedule serializable
- A conflict serializable schedule

T1	T2
R(A)	
	Commit
W(A)	
Commit	

Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

- Timestamp CC can be modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.

Recoverability

- If TS(T1) = 1 and TS (T2)= 2 the schedule is permitted by timestamp protocol (TSP) with or without TRL
- The TSP can be modified to disallow such schedules by buffering all write actions until the transaction commits
- When T1 wants to write to A, WTS(A) is updated to reflect this action, but the change to A is not carried out immediately instead it is recorded in a private workspace (buffer)
- When T2 wants to read A, its timestamp is compared with WTS(A), and the read is seen to be permissible
- T2 is blocked till T1 completes
- If T1 commits, the change to A is copied from the buffer, otherwise the changes in the buffer are discarded
- T2 is then allowed to read A

Multiversion Timestamp Protocol

- Multiversion schemes keep old versions of data item to increase concurrency.
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- when a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to $TS(T_i)$.
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
  2. If transaction T_i issues a **write**(Q)
     1. if $TS(T_i)$ < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if $TS(T_i)$ = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. TS(Ti) > R-timestamp (Qk), a new version of Q is created.
- Observe that
  - Reads always succeed.
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability.

MVCC: Implementation Issues

- Reading of data items also requires the updating of R-timestamp field (2 disk accesses)
- Conflicts are resolved though rollbacks rather than through waits (expensive)
- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Qk and Qj, and both versions have W-timestamp less than the timestamp of the oldest transaction in the system. Then the older of the 2 versions (Qk, Qj) will not be used again and can be deleted

  - the oldest active transaction has timestamp > 9, then Q5 will never be required again

Multiversion CC

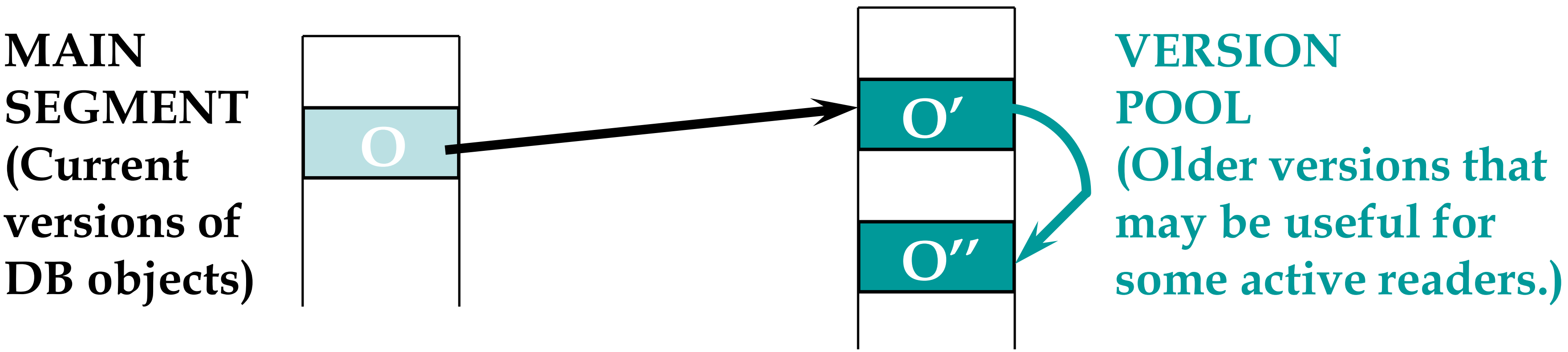
- The goal is to ensure that a transaction never has to wait to read
- Maintain several versions of each database object, each with a write timestamp, and let transaction Ti read the most recent version whose timestamp precedes TS(Ti)
- If transaction Ti wants to write to object, ensure that the object has not already been read by some other transaction Tj such that TS(Ti)<TS(Tj), if we allow Ti to write to such an object, its change should be seen by Tj for serializability, but Tj which read the object at some time in the past, will not see Ti’s change

MVCC

- Every object has a read timestamp
- Whenever a transaction reads the object, the read timestamp is set to the maximum of the current read timestamp and the reader's timestamp
- If Ti wants to write an object O and TS(Ti)< RTS(O), Ti is aborted and restarted with a new larger timestamp
- Otherwise Ti creates a version of O and sets the read and write timestamps of the new version to TS(Ti)
- Reads are never blocked but there is overhead of maintaining the versions

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:

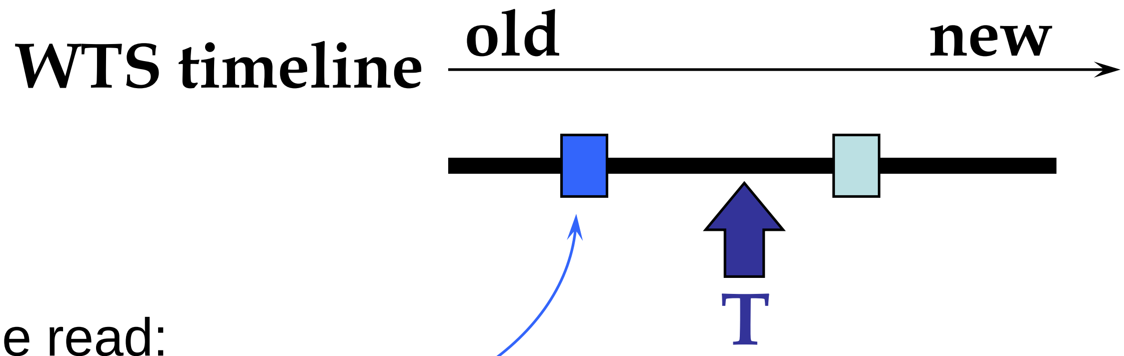


- Readers are always allowed to proceed.
  - But may be blocked until writer commits.

Multiversion CC ….

- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".
- Each Xact is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.

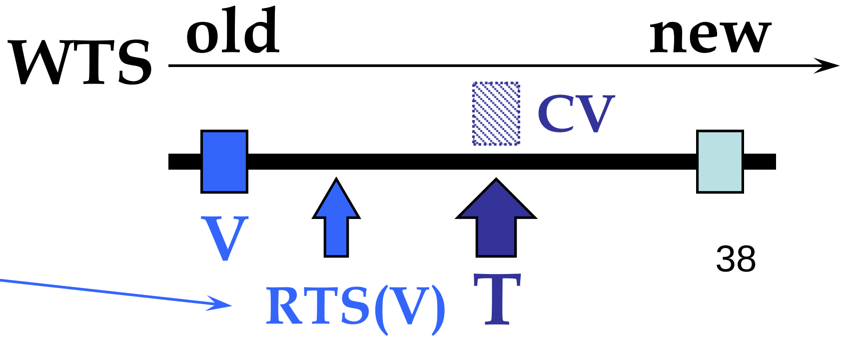
Reader Xact



- 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.)
- Assuming that some version of every object exists from the beginning of time, Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If 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 until T commits; other Xacts can see TS values but can't read version CV.)
  - Else, reject write.



Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem

Summary (Contd.)

- Index locking is common, and affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Bayer-Schkolnick illustrates potential for improvement.
- In practice, better techniques now known; do record-level, rather than page-level locking.

Summary (Contd.)

- Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages); should not be confused with tree index locking!
- Optimistic CC aims to minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.
- SQL-92 provides different isolation levels that control the degree of concurrency

Summary (Contd.)

- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Xacts, which is similar to locking.
- Multiversion Timestamp CC is a variant which ensures that read-only Xacts are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.