Database Management Systems

Concurrency Control

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
and
Centre for Artificial Intelligence and Machine Learning
Indian Statistical Institute, Kolkata

June, 2021

Basics

Concurrency control is the way to preserve isolation of transactions while managing concurrent execution

Assumption: No failure occurs during concurrent execution.

We know that serializability ensures the consistency of a database.

So, concurrency control schemes are mostly based on the serializability property.

Note: Serializable concurrency control might have adverse effects on long-duration transactions.

Lock-based protocols – Basics

A lock is a mechanism to control concurrent access to a data item in a mutually exclusive manner

The two most common lock modes are:

- Exclusive (**X**) – Data item can be both read as well as written
- Shared (**S**) – Data item can only be read

Lock requests are made to the concurrency control manager and a transaction can proceed only after its *request* is granted.

Note: A lock held by a transaction on an item may be granted another lock requested by another transaction.

Lock-based protocols – Basics

Definition (Lock compatibility)

If a transaction can be granted a lock A on an item immediately, in spite of the presence of another lock B on the same data item, then it is said that A is compatible with B.

The lock compatibility relations:

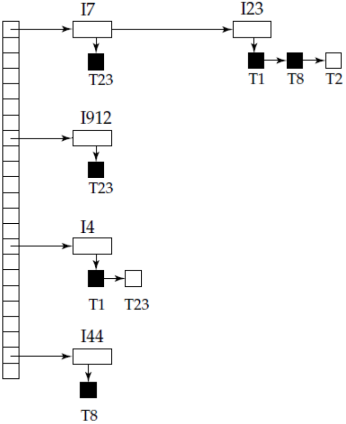
	S	X
S	True	False
X	False	False

Definition (Locking protocol)

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Implementation of locking

The lock manager uses a hash table indexed on the name of a data item. It finds the linked list, in the order in which the requests arrived, for a currently locked data item.



A lock table

- A blackbox denotes an approved lock request
- A whitebox denotes a waiting lock request
- The type of lock gets recorded
- New lock requests get added to the end of the queue
- Unlock requests or abort deletes the corresponding requests

Lock-based protocols – Managing serializability

The following protocol does not guarantee serilizability:

Transaction T_1	
lock-S(ISI$_{PC}$)	
read (ISI$_{PC}$)	
unlock(ISI$_{PC}$)	Updates on IISc$_{PC}$ is not admissible!!!
lock-S(IISc$_{PC}$)	Updates on IISc$_{PC}$ is not admissible!!!
read (IISc$_{PC}$)	
unlock(IISc$_{PC}$)	
display(ISI$_{PC}$+IISc$_{PC}$)	

Note: If IISc$_{PC}$ (or ISI$_{PC}$) gets updated in-between the reads of ISI$_{PC}$ and IISc$_{PC}$ (or IISc$_{PC}$ and ISI$_{PC}$), then the sum will be displayed wrong.

Lock-based protocols – Managing serializability

The following protocol guarantees serilizability:

Transaction T_1
lock-S(ISI$_{PC}$)
read (ISI$_{PC}$)
lock-S(IISc$_{PC}$)
read (IISc$_{PC}$)
display(ISI$_{PC}$+IISc$_{PC}$)
unlock(ISI$_{PC}$)
unlock(IISc$_{PC}$)

Lock-based protocols – Drawbacks

Transaction T_1	Transaction T_2
lock-X(ISI$_{PC}$)	
read(ISI$_{PC}$)	
ISI$_{PC}$ $\leftarrow$ ISI$_{PC}$ - 10	
write(ISI$_{PC}$)	
	lock-S(IISc$_{PC}$)
	read(IISc$_{PC}$)
	lock-S(ISI$_{PC}$)
lock-X(IISc$_{PC}$)	

Deadlock – lock-S(ISI$_{PC}$) causes T_2 to wait for T_1 to release its lock on ISI$_{PC}$, whereas lock-X(IISc$_{PC}$) causes T_1 to wait for T_2 to release its lock on $IISc_{PC}$.

Solution: T_1 or T_2 must be rolled back and the corresponding lock should be released.

Lock-based protocols – Drawbacks

Transaction T_1	Transaction T_2	Transaction T_3
lock-X(IISc$_{PC}$)		
lock-S(ISI$_{PC}$)		
read(ISI$_{PC}$)		
ISI$_{PC}$ $\leftarrow$ ISI$_{PC}$ - 10		
		lock-S(ISI$_{PC}$)
write(IISc$_{PC}$)		
		read(ISI$_{PC}$)
	lock-X(ISI$_{PC}$)	

Starvation – lock-X(ISI$_{PC}$) causes T_2 to wait for both T_1 and T_3 to release their locks on ISI$_{PC}$, and T_2 is repeatedly rolled back due to deadlocks.

Solution: Concurrency control manager should be designed appropriately.

Two-phase locking protocols – Basics

Working principle:

1. **Phase 1 (Grow)** – A transaction may obtain locks, but may not release any lock.
2. **Phase 2 (Shrink)** – A transaction may release locks, but may not obtain any new locks.

Two-phase locking protocols ensure conflict serializability.

Note: The serialization is determined based on the order of transaction *lock points* (where a transaction acquires its final lock).

Two-phase locking protocols – Implementation

Two-phase locking with lock conversions:

- **Phase 1**
  - can acquire a lock-S on the data item
  - can acquire a lock-X on the data item
  - can convert a lock-S to a lock-X (upgrade)
- **Phase 2**
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

Two-phase locking protocols – An example

Transaction T_1	Transaction T_2
lock-S(IISc$_{PC}$)	
	lock-S(IISc$_{PC}$)
lock-S(ISI$_{PC}$)	
	lock-S(ISI$_{PC}$)
lock-S(IITK$_{PC}$)	
lock-S(IITD$_{PC}$)	
	unlock(IISc$_{PC}$)
	unlock(ISI$_{PC}$)
lock-S(IITB$_{PC}$)	
upgrade(IISc$_{PC}$)	
write(IISc$_{PC}$)	

Note: Avoiding lock-X on IISc$_{PC}$ at the beginning provides more concurrency to schedules. The lock can be upgraded as and when required (not via unlock followed by a lock-X).

Two-phase locking protocols – Drawbacks

Deadlock: In two-phase locking protocol, two transactions might wait for each other to release their corresponding locks on two different items.

Solution: Rollback any of the transactions causing the deadlock.

Cascading rollback: A single transaction failure leads to a series of transaction rollbacks.

Solution: Either use *strict two-phase locking protocol* (a transaction must hold all its exclusive locks till it commits/aborts) or *rigorous two-phase locking protocol* (all locks are held till commit/abort).

Dirty reads

A *dirty read* (or *uncommitted dependency*) occurs when a transaction is allowed to read a data item that has been updated by another running transaction and not yet committed. It causes cascading rollback (rollback in T_1 causes rollbacks in T_2, T_3).

Transaction T_1	**Transaction T_2**	**Transaction T_3**
	lock-X($IISc_{PC}$)	
lock-X(ISI_{PC})		
read(ISI_{PC})		
$ISI_{PC} \leftarrow ISI_{PC}$ - 10		
write(ISI_{PC}) ↑ *rollback*		
unlock(ISI_{PC})		
	lock-X(ISI_{PC})	
	read(ISI_{PC})	
	write(ISI_{PC})	
	unlock(ISI_{PC})	
		lock-S(ISI_{PC})
		read(ISI_{PC})

Insertion and deletion under two-phase locking

It can be used with two-phase locking protocol.

Working principle:

1. A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
2. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.

Insertion and deletion under two-phase locking – Drawback

Phantom phenomenon: A transaction that scans a relation and a transaction that inserts a tuple in the relation might conflict in spite of not accessing any tuple in common.

Solution: Associate a data item with the relation to represent the information about what tuples the relation contains.

Graph-based protocols – Basics

Working principle:

1. Graph-based protocols impose a partial ordering $\rightarrow$ on the set of all items $I = I_1, I_2, ..., I_n$.
2. It also includes the constraint that if $I_i \rightarrow I_j$ then any transaction accessing both I_i and I_j must access I_i before accessing I_j.

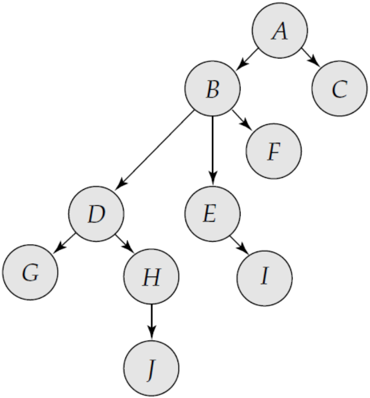
It implies that the set I may now be viewed as a directed acyclic graph that is known as database graph.

Graph-based protocols – An example

Tree protocol:

- Only exclusive locks are allowed.
- The first lock by T_i may be on any item. Subsequently, an item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

Graph-based protocols – Visualization



Visualizing a tree protocol

Timestamp-based protocols – Basics

In concurrency control, timestamps are implemented either with the *system clock* or using a *logical counter*.

Working principle:

1. Each transaction (say T_i) obtains a timestamp (say $\mathcal{TS}(T_i)$) on entering the system.
2. If an old transaction T_i has timestamp $\mathcal{TS}(T_i)$, a new transaction T_j is assigned a timestamp $\mathcal{TS}(T_j)$ such that $\mathcal{TS}(T_i) < \mathcal{TS}(T_j)$.

This ensures concurrent execution and the timestamps determine the serializability order.

Implementation schemes:

1. W-timestamp(Q) – The timestamp of a transaction that has executed the last write(Q) successfully.
2. R-timestamp(Q) – The timestamp of a transaction that has executed the last read(Q) successfully.

Timestamp-based protocols – Implementation

Timestamp-ordering protocol:

if Transaction T_i issues read(Q) **then**
    if $\mathcal{TS}(T_i) <$ W-timestamp(Q) **then**
        Reject read(Q) and roll back T_i. // T_i needs to read a value of Q already overwritten
    else
        Execute read(Q) and set R-timestamp(Q) $=$ max{R-timestamp(Q), $\mathcal{TS}(T_i)$}.
    end if
end if
if Transaction T_i issues write(Q) **then**
    if $\mathcal{TS}(T_i) <$ R-timestamp(Q) **then**
        Reject write(Q) and roll back T_i. // The value of Q that T_i is producing was needed previously, so it is assumed that it would never be produced
    end if
    If $\mathcal{TS}(T_i) <$ W-timestamp(Q), reject write(Q) and roll back T_i. // T_i is attempting to write an obsolete value of Q
    Otherwise, execute the write operation and set W-timestamp(Q) $= \mathcal{TS}(T_i)$.
end if

Note: Transactions arriving earlier cannot read/write later.

Timestamp-based protocols – Example I

See below an implementation of the timestamp-ordering protocol on five transactions (T_1, T_2, T_3, T_4 and T_5) having timestamps 3, 2, 4, 10 and 1, respectively.

T_1	T_2	T_3	T_4	T_5
				read(ISI_{PC})
	read($IISc_{PC}$)			
read($IISc_{PC}$)				
		write($IISc_{PC}$)		
		write(ISI_{PC})		
				read(ISI_{PC})
	read(ISI_{PC})			
	abort			
read(ISI_{PC})				
			write($IISc_{PC}$)	
		write($IISc_{PC}$)		
		commit		
				write($IISc_{PC}$)
				write(ISI_{PC})

Timestamp-based protocols – Revised implementation

Thomas' write rule:

if Transaction T_i issues read(Q) **then**
    if $\mathcal{TS}(T_i) <$ W-timestamp(Q) **then**
        Reject read(Q) and roll back T_i.
    else
        Execute read(Q) and set R-timestamp(Q) $=$ max{R-timestamp(Q), $\mathcal{TS}(T_i)$}.
    end if
end if
if Transaction T_i issues write(Q) **then**
    if $\mathcal{TS}(T_i) <$ R-timestamp(Q) **then**
        Reject write(Q) and roll back T_i.
    end if
    If $\mathcal{TS}(T_i) <$ W-timestamp(Q), ignore write(Q). // T_i is not rolled back*
    Otherwise, execute the write operation and set W-timestamp(Q) $= \mathcal{TS}(T_i)$.
end if

*It ensures view serializability for schedules that are not conflict serializable.

Timestamp-based protocols – Advantages and drawbacks

Serializability guaranteed: Timestamp-ordering protocol ensures serializability since all the arcs in the precedence graph do not form any cycle in the precedence graph.

Freedom from deadlock: Timestamp-ordering protocol ensures freedom from deadlock because no transaction ever waits.

Cascading rollback problem: A single transaction failure leads to a series of transaction rollbacks.

Recoverability problem: A transaction may not be recoverable.

Validation-based protocols – Basics

It is also called *optimistic concurrency control* since transaction executes fully in the hope that all will go well during validation.

Working principle:

1. **Read and execution phase** – Transaction T_i writes only to temporary local variables.
2. **Validation phase** – Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
3. **Write phase** – If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

Each transaction must go through the three aforementioned phases in the same order.

Validation-based protocols – Basics

Implementation schemes:

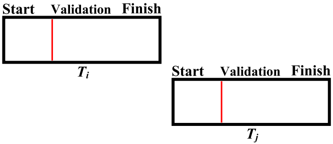
1. Timestamp Start(T_i) – The time when T_i started its execution
2. Timestamp Validation(T_i) – The time when T_i entered its validation phase
3. Timestamp Finish(T_i) – The time when T_i finished its write phase

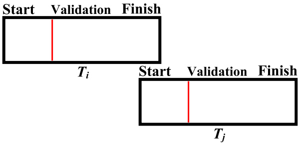
to increase concurrency, serializability order is determined by the timestamp given at validation time i.e. $\mathcal{TS}(T_i)$ is set to Validation(T_i).

Validation-based protocols – An example

Validation test: To ensure one of the following things:

- There is no overlapped execution
- Writes of T_i and T_j do not affect reads of T_j and T_i, respectively.





for T_j with $\mathcal{TS}(T_i) < \mathcal{TS}(T_j)$ **do**
    if (Finish(T_i) $<$ Start(T_j)) or (Start(T_j) $<$ Finish(T_i) $<$ Validation(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j) **then**
        Commit T_j.
    else
        Abort T_j.
    end if
end for

Validation-based protocols – An example

Transaction T_1	**Transaction** T_2
read(IISc$_{PC}$)	
	read(IISc$_{PC}$)
	IISc$_{PC}$ $\leftarrow$ IISc$_{PC}$ - 10
	read(ISI$_{PC}$)
	ISI$_{PC}$ $\leftarrow$ ISI$_{PC}$ $+$ 10
read(ISI$_{PC}$)	
$<$ Validate $>$	
display(ISI$_{PC}$ $+$ IISc$_{PC}$)	
	write(IISc$_{PC}$)
	write(ISI$_{PC}$)

Multiple granularity – Basics

It allows data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.

Working principle:

1. It is represented graphically as a tree.
2. When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendants in the same mode.

Granularity of locking can be at two levels:

- Fine granularity (lower in tree) – ensures high concurrency and locking overhead
- Coarse granularity (higher in tree) – ensures low concurrency and locking overhead.

Multiple granularity – Basics

Different locking modes:

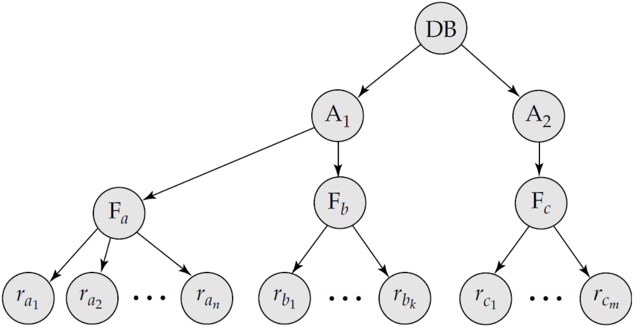
- **IS** – Intention-shared lock that indicates explicit locking at a lower level of the tree but only with shared locks.
- **IX** – Intention-exclusive lock that indicates explicit locking at a lower level with exclusive or shared locks.
- **S** – Shared lock as used conventionally.
- **SIX** – Shared and intention-exclusive lock in which the root node (of the subtree) is S-locked and explicit locking is being done at a lower level with exclusive locks.
- **X** – Exclusive lock as used conventionally.

Multiple granularity – Basics

The lock compatibility relations:

	IS	IX	S	SIX	X
IS	True	True	True	True	False
IX	True	True	False	False	False
S	True	False	True	False	False
SIX	True	False	False	False	False
X	False	False	False	False	False

Multiple granularity – Visualization



Hierarchy of granularity

Levels from the top to bottom: database (DB), area (A_1, A_2), file (F_a, F_b, F_c) and record ($r_{a_1}, r_{a_2}, \ldots, r_{a_n}, r_{b_1}, \ldots, r_{b_k}, r_{c_1}, \ldots, r_{c_m}$)

Multiple granularity – Implementation

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node i.e. T_i is two-phase.
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Multiversion schemes – Timestamp ordering

Each data item Q has a sequence of versions $< Q_1, Q_2, \ldots, Q_m >$.
Each version Q_k contains three data fields:

- Content – The value of version Q_k.
- W-timestamp(Q_k) – The timestamp of the transaction that wrote (created) version Q_k.
- R-timestamp(Q_k) – The largest timestamp of a transaction that successfully read version Q_k.

Working principle:

1. When a transaction T_i creates a new version Q_k of Q, set W-timestamp(Q_k) $= \mathcal{TS}(T_i)$ and R-timestamp(Q_k) $= \mathcal{TS}(T_i)$.
2. Update R-timestamp(Q_k) with $\mathcal{TS}(T_j)$ whenever a transaction T_j reads Q_k, and $\mathcal{TS}(T_j) >$ R-timestamp(Q_k).

Multiversion schemes – Two-phase Locking

Differentiates between read-only transactions and update transactions.

Working principle:

1. Update transactions acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
2. Read-only transactions are assigned a timestamp by reading the current value of *timestamp counter* before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Concurrency in indexes – Basics

This approach can solve the phantom phenomenon.

Working principle:

1. Every relation must have at least one index.
2. A transaction can access tuples only after finding them through one or more indices on the relation.
3. A transaction T_i that performs a read (lookup) must lock all the index leaf nodes that it accesses in shared mode, even if the leaf node does not contain any tuple satisfying the index lookup.
4. A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r must update all indices to r and must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete.
5. The rules of the two-phase locking protocol must be observed.