

1 Concurrency Control Protocols

- Lock-based Protocols
- Graph-based Protocols
- Timestamp-based Protocols
- Validation-based Protocols

2 Multiple Granularity

3 Multiversion Schemes

4 Concurrency in Indexes

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:

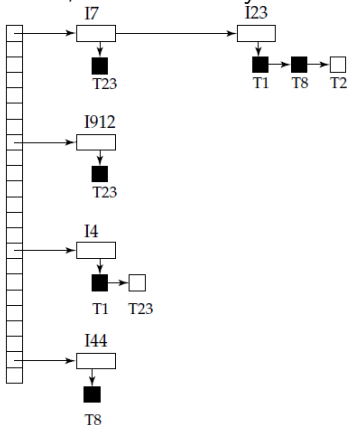
	S	X
S	True	False
X	False	False

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 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

A lock table

Lock-based protocols – Managing serializability

The following protocol does not guarantee serializability:

Transaction T_1
lock-S(ISI_{PC})
read (ISI_{PC})
unlock(ISI_{PC})
lock-S($IISc_{PC}$)
read ($IISc_{PC}$)
unlock($IISc_{PC}$)
display($ISI_{PC} + IISc_{PC}$)

Updates on $IISc_{PC}$ is not admissible!!!

Updates on $IISc_{PC}$ is not admissible!!!

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 serializability:

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

Deadlock – $\text{lock-S}(\text{ISI}_{PC})$ causes T_2 to wait for T_1 to release its lock on ISI_{PC} , whereas $\text{lock-X}(\text{IISc}_{PC})$ 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
$\text{lock-X}(\text{ISc}_{PC})$ $\text{lock-S}(\text{ISI}_{PC})$ $\text{read}(\text{ISI}_{PC})$ $\text{ISI}_{PC} \leftarrow \text{ISI}_{PC} - 10$ $\text{write}(\text{ISc}_{PC})$	 $\text{lock-X}(\text{ISI}_{PC})$	 $\text{lock-S}(\text{ISI}_{PC})$ $\text{read}(\text{ISI}_{PC})$

Starvation – $\text{lock-X}(\text{ISI}_{PC})$ 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(ISI _{PC})	lock-S(IISc _{PC})
lock-S(IITK _{PC})	lock-S(ISI _{PC})
lock-S(IITD _{PC})	
	unlock(IISc _{PC})
lock-S(IITB _{PC})	unlock(ISI _{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(ISI_{PC}) read(ISI_{PC}) $ISI_{PC} \leftarrow ISI_{PC} - 10$ write(ISI_{PC}) $\uparrow$ <i>rollback</i> unlock(ISI_{PC})	lock-X($IISc_{PC}$) lock-X(ISI_{PC}) <i>read(ISI_{PC})</i> write(ISI_{PC}) unlock(ISI_{PC})	lock-S(ISI_{PC}) <i>read(ISI_{PC})</i>

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 $I = I_1, I_2, \dots, I_n$.
- 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 .

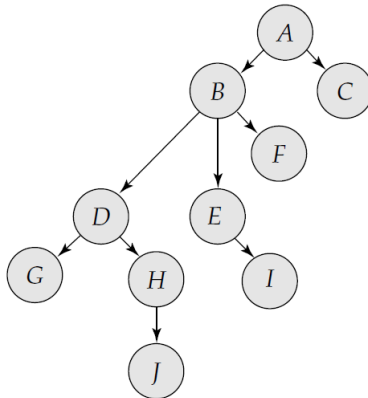
It implies that the set I may now be viewed as a directed acyclic graph that is known as database graph.

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 $TS(T_i)$) on entering the system.
- 2 If an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned a timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.

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:

- 1: **if** Transaction T_i issues read(Q) **then**
- 2: **if** $TS(T_i) < W\text{-timestamp}(Q)$ **then**
- 3: Reject read(Q) and roll back T_i . // T_i needs to read a value of Q already overwritten
- 4: **else**
- 5: Execute read(Q) and set $R\text{-timestamp}(Q) = \max\{R\text{-timestamp}(Q), TS(T_i)\}$.
- 6: **end if**
- 7: **end if**
- 8: **if** Transaction T_i issues write(Q) **then**
- 9: **if** $TS(T_i) < R\text{-timestamp}(Q)$ **then**
- 10: 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
- 11: **end if**
- 12: **If** $TS(T_i) < W\text{-timestamp}(Q)$, reject write(Q) and roll back T_i . // T_i is attempting to write an obsolete value of Q
- 13: Otherwise, execute the write operation and set $W\text{-timestamp}(Q) = TS(T_i)$.
- 14: **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(IISc _{PC})	read(IISc _{PC})	write(IISc _{PC}) write(ISl _{PC})		read(ISl _{PC})
read(ISl _{PC})	read(ISl _{PC}) abort	write(IISc _{PC}) commit	write(IISc _{PC})	read(ISl _{PC})
				write(IISc _{PC}) write(ISl _{PC})

Timestamp-based protocols – Revised implementation

Thomas' write rule:

- 1: **if** Transaction T_i issues read(Q) **then**
- 2: **if** $TS(T_i) < W\text{-timestamp}(Q)$ **then**
- 3: Reject read(Q) and roll back T_i .
- 4: **else**
- 5: Execute read(Q) and set $R\text{-timestamp}(Q) = \max\{R\text{-timestamp}(Q), TS(T_i)\}$.
- 6: **end if**
- 7: **end if**
- 8: **if** Transaction T_i issues write(Q) **then**
- 9: **if** $TS(T_i) < R\text{-timestamp}(Q)$ **then**
- 10: Reject write(Q) and roll back T_i .
- 11: **end if**
- 12: **If** $TS(T_i) < W\text{-timestamp}(Q)$, ignore write(Q). // T_i is not rolled back*
- 13: Otherwise, execute the write operation and set $W\text{-timestamp}(Q) = TS(T_i)$.
- 14: **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. $TS(T_i)$ is set to Validation(T_i).

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:

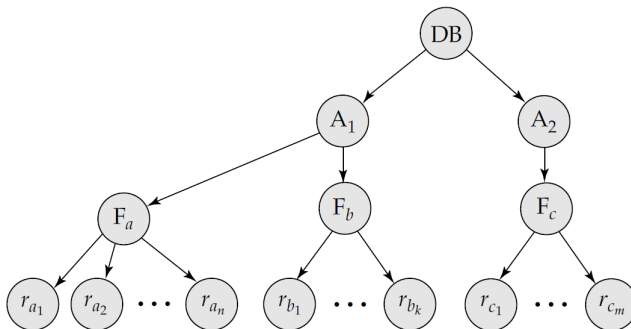
- Fine granularity (lower in tree) – ensures high concurrency and locking overhead
- Coarse granularity (higher in tree) – ensures low concurrency and locking overhead.

- **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.

Figure 1

	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 ($r_{a_1}, r_{a_2}, \dots, r_{a_n}, r_{b_1}, \dots, r_{b_k}, r_{c_1}, \dots, r_{c_m}$)

Each version Q_k contains three data fields:

- ### Working principle:

- ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ◻ ↺ 🔍 ↻

This approach can solve the phantom phenomenon.

- 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.