Concurrency Control

COSC 404 – Database System Implementation

Concurrency Control Overview

Concurrency control (CC) is a mechanism for guaranteeing that concurrent transactions in the database exhibit the ACID properties. Specifically, the isolation property.

There are different concurrency control protocols:

- lock-based protocols
- timestamp protocols
- validation protocols
- snapshot isolation

Lock-Based Protocols

A ***lock*** is a mechanism to control concurrent access to data.

- An item can only be accessed through the lock.

Data items can be locked in two modes:

- ***exclusive (X) mode***: Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
- ***shared (S) mode***: Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to the concurrency control manager. A transaction can only proceed after the request is *granted* and must follow the restrictions of the lock.

Lock-Based Protocols (2)

Lock-compatibility matrix:

	S	X
S	true	false
X	false	false

A transaction may be granted a lock on an item if the requested lock is *compatible* with locks already held on the item by other transactions.

- Any # of transactions can hold shared locks on an item.
- If any transaction holds an exclusive lock on the item, no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to ***wait*** until all incompatible locks held by other transactions are released. The lock is then granted.

Lock-Based Protocol Example

Example of a transaction performing locking:

lock-S(*A*);

read (*A*);

unlock(*A*); ⇐ Another transaction updates B here.

lock-S(*B*);

read (*B*);

unlock(*B*);

display(*A+B*)

Simple locking is not sufficient to guarantee serializability.

- If *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum is wrong.
- 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.

Pitfalls of Lock-Based Protocols

Consider the partial schedule:

T_3	T_4
lock-X(*B*)	
read(*B*)	
B:- B-50	
write(*B*)	
	lock-S(*A*)
	read(*A*)
	lock-S(*B*)
lock-X(*A*)	

- Neither T_3 nor T_4 can make progress as executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a ***deadlock***. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (2)

The potential for deadlock exists in most locking protocols.

Starvation is also possible if the concurrency control manager is badly designed. Examples:

- A transaction may be waiting for an exclusive lock on an item, while a sequence of other transactions request and are granted a shared lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.

The concurrency control manager can be designed to prevent starvation.

- For example, do not grant a shared lock if the item is exclusively locked or a transaction is waiting for a lock-X.

Locking Question

Question: Which of the following statements are true?

A) A shared lock allows a transaction to write a data item.

B) More than one transaction can have a shared lock on an item.

C) More than one transaction can have an exclusive lock on an item.

D) Deadlock can always be avoided by releasing locks as early as possible.

E) More than one statement is true.

The Two-Phase Locking Protocol

Two-Phase Locking (***2PL***) ensures conflict-serializable schedules by requiring all locks be acquired before first unlock.

Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

The protocol ensures serializability. It can be proved that the transactions can be serialized in the order of their *lock points* (i.e. the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (2)

2PL *does not* ensure freedom from deadlocks.

- Cascading roll-back is also possible under two-phase locking.

Conservative 2PL is deadlock free as all locks must be pre-declared and allocated at transaction start time.

Strict 2PL prevents cascading rollback as a transaction holds all its exclusive locks until it commits/aborts.

- Thus, uncommitted data is locked and cannot be accessed.

Rigorous 2PL is even stricter as *all* locks are held till commit/abort. (also cascade free)

- Transactions can be serialized in the order that they commit.
- Both strict and rigorous 2PL produce strict schedules. Database systems that use locking use strict or rigorous 2PL.

Lock Conversions

Increased concurrency is possible by allowing lock conversions.

- **Upgrade** - convert shared lock to exclusive lock
- **Downgrade** - convert exclusive lock to shared lock

For two-phase locking with lock conversions:

- Upgrades and lock acquires are allowed in growing phase.
- Downgrades and lock releases are in the shrinking phase.

Automatic Acquisition of Locks

A simple automated algorithm can place lock requests for a transaction T_i issuing the standard read/write instructions:

- The operation read(*D*) is processed as:
  - if T_i has a lock on *D* then read(*D*) otherwise
  - request a **lock-S** on *D* (may be necessary to wait for a **lock-X**)
  - when **lock-S** request is granted, then read(*D*)
- The operation write*(D)* is processed as:
  - if T_i has a **lock-X** on *D* then write(*D*) otherwise
  - if T_i has a **lock-S** on *D* then upgrade lock on D to **lock-X**
    - may have to wait for upgrade
  - otherwise request a new **lock-X**
  - finally write(*D*) when receive upgrade or new lock
- All locks are released after commit or abort.

Example on Auto Lock Insertion

Abbreviations:

- A transaction T_i requesting a **lock-S** on D is given as: $sl_i(D)$.
- A transaction T_i requesting a **lock-X** on D is given as: $xl_i(D)$.
- A transaction T_i unlocking a data item D is given as: $ul_i(D)$.

Given transaction T_1, insert lock operations according to 2PL:

T_1: $r_1(A)$; $r_1(C)$; $w_1(B)$; $w_1(C)$;

locks may be released anytime after this operation when not needed

Basic 2PL:

$sl_1(A)$; $r_1(A)$; $sl_1(C)$; $r_1(C)$; $xl_1(B)$; $w_1(B)$; $xl_1(C)$; $ul_1(A)$; $ul_1(B)$; $w_1(C)$; $ul_1(C)$; c_1;

Example on Auto Lock Insertion (2)

Conservative 2PL:

$atomic(sl_1(A), xl_1(C), xl_1(B))\ r_1(A);\ ul_1(A);\ r_1(C);\ w_1(B);\ ul_1(B);\ w_1(C);\ ul_1(C);\ c_1;$

locks may be released after they are no longer needed

Strict 2PL:

$sl_1(A);\ r_1(A);\ xl_1(C);\ r_1(C);\ xl_1(B);\ w_1(B);\ xl_1(C);\ ul_1(A);\ w_1(C);\ c_1;\ ul_1(B);\ ul_1(C);$

read locks may be released before commit (after last lock operation)

Rigorous 2PL:

$sl_1(A);\ r_1(A);\ xl_1(C);\ r_1(C);\ xl_1(B);\ w_1(B);\ );\ xl_1(C);\ w_1(C);\ c_1;\ ul_1(A);\ ul_1(B);\ ul_1(C);$

all locks released after commit

2PL Question

Question: How many of the following statements are true?

- i) Conservative 2PL is deadlock-free.
- ii) Rigorous 2PL releases only write locks after commit.
- iii) Lock upgrades are allowed during the shrinking phase of 2PL.
- iv) Strict 2PL produces strict schedules.

A) 0

B) 1

C) 2

D) 3

E) 4

Questions on 2PL

1) Given the following transactions, insert lock operations according to 2PL:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$;

2) Write one non-serial 2PL schedule or argue why one is not possible.

3) Repeat #1 and #2 for these transactions:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; c_1

T_2: $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$; $r_2(B)$; $w_2(B)$; c_2

T_3: $r_3(C)$; $r_3(A)$; $\mathrm{w}_3(\mathrm{C})$; c_3

Multiple Granularity

To this point, we have been locking individual data items. It is beneficial to allow locking of various sized data items.

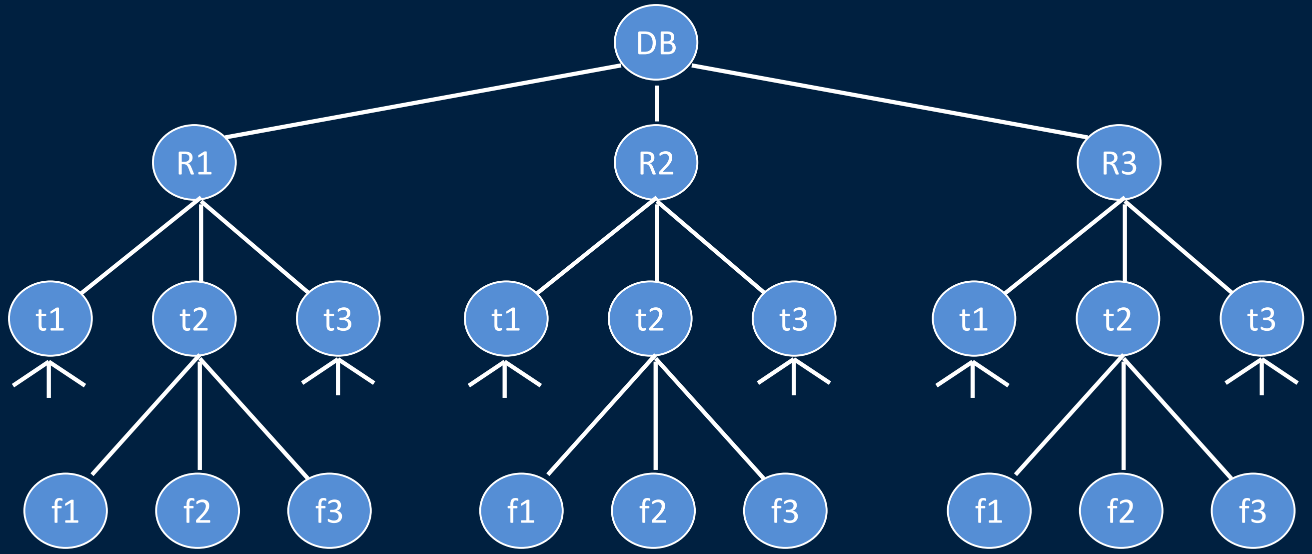
- Define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree.

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 (level in tree where locking is done):

- *fine granularity* (lower in tree): high concurrency, high locking overhead (e.g. record locking, attribute locking)
- *coarse granularity* (higher in tree): low locking overhead, low concurrency (e.g. table locking, database locking)

Example of Granularity Hierarchy



The highest level in the hierarchy is the entire database.

The levels below are *relation, tuple* and *field* in that order.

Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

- ***intention-shared* (IS):** indicates explicit locking at a lower level of the tree but only with shared locks.
- ***intention-exclusive* (IX):** indicates explicit locking at a lower level with exclusive or shared locks
- ***shared and intention-exclusive* (SIX):** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

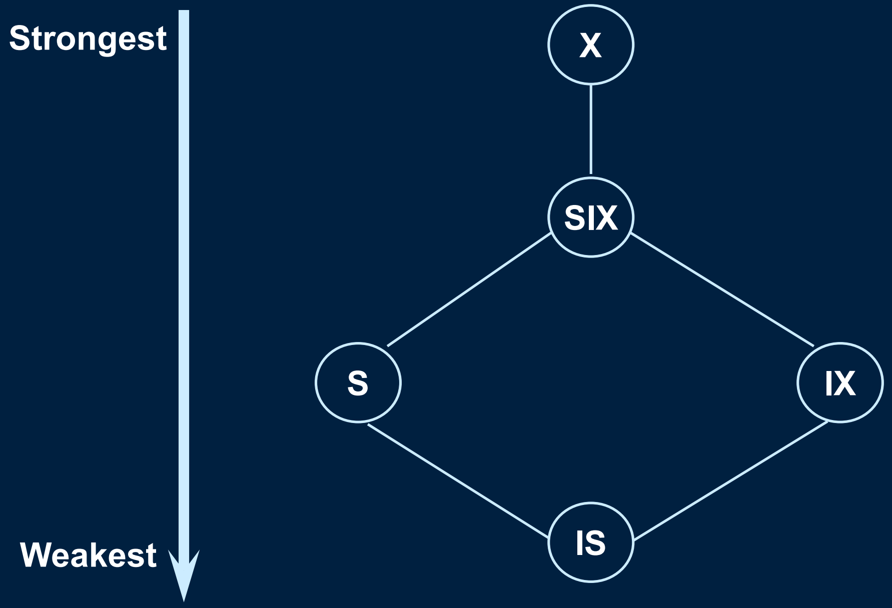
Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

Multi Granularity Lock "Strength"



Multiple Granularity Locking

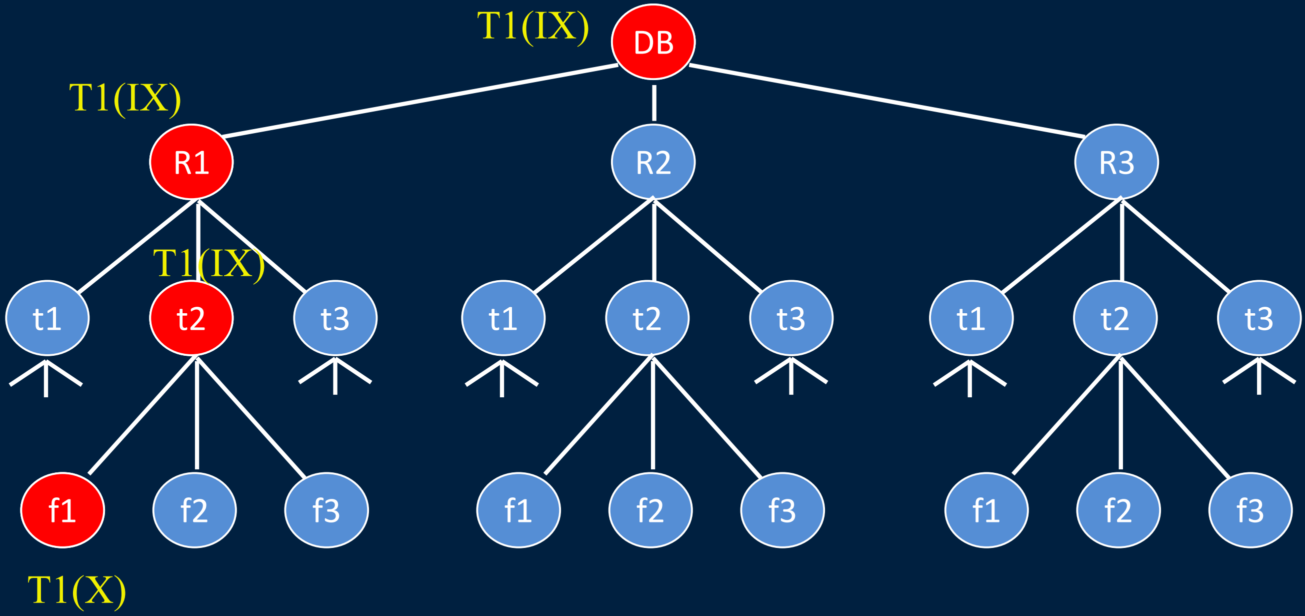
Transaction T_i can lock a node Q using the rules:

- The lock compatibility matrix must be observed.
- The root of the tree must be locked first (in any mode).
- 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.
- 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.
- T_i can lock a node only if it has not previously unlocked any node (that is, this is a variant of two-phase locking).
- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Protocol is not deadlock-free.

Locks are acquired in root-to-leaf order, and released in leaf-to-root order.

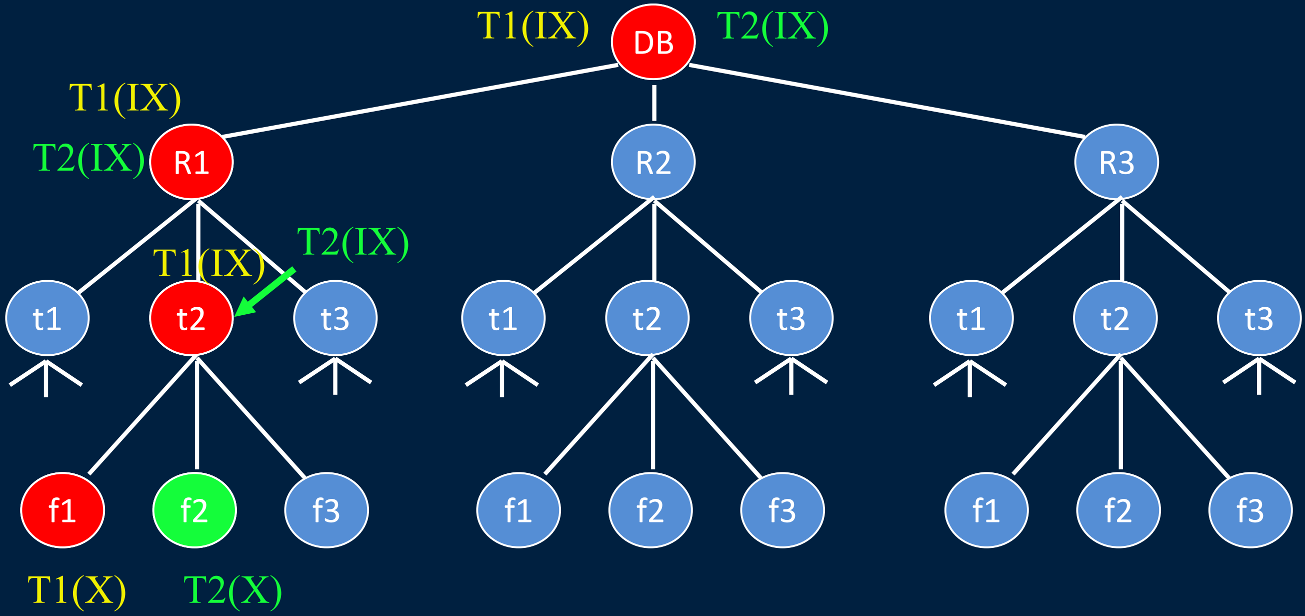
Multiple Granularity Locking Example

T1 wants to lock R1.t2.f1 in X-mode.



Multiple Granularity Locking Example (2)

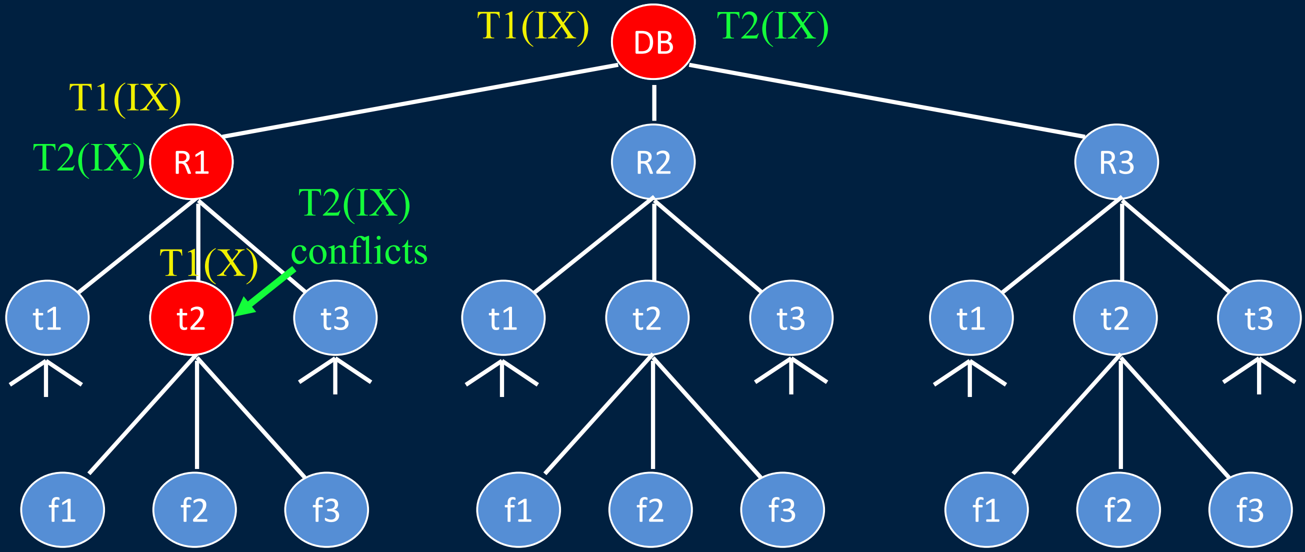
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (3)

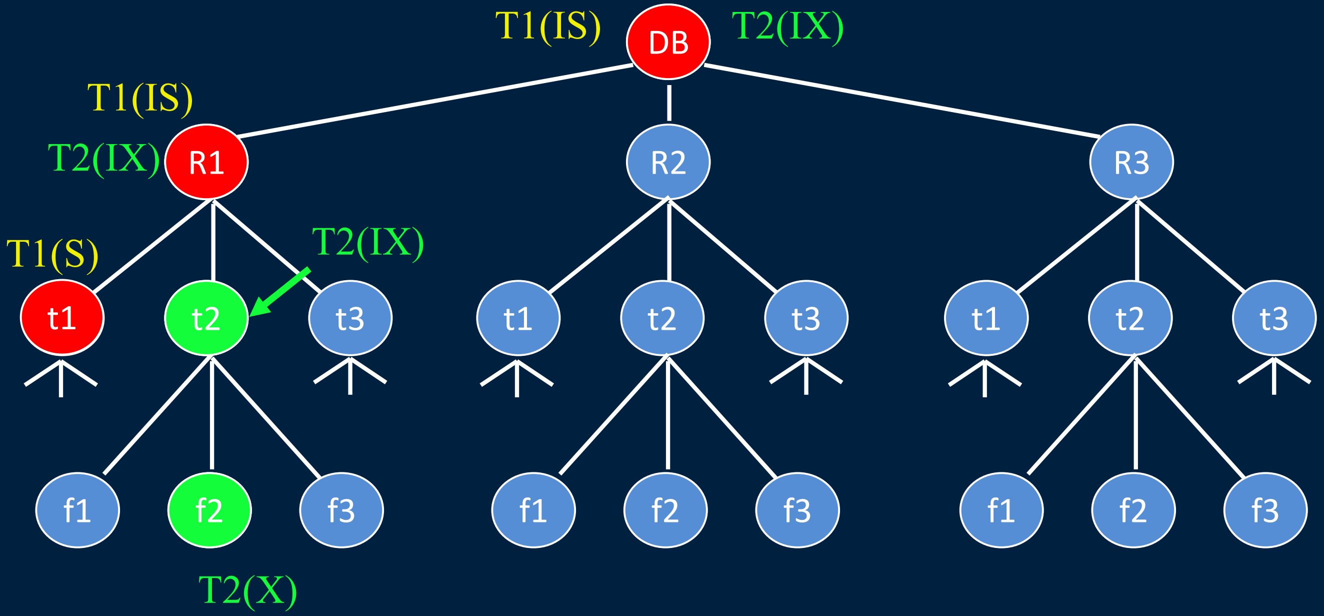
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



No, conflict at t2!

Multiple Granularity Locking Example (4)

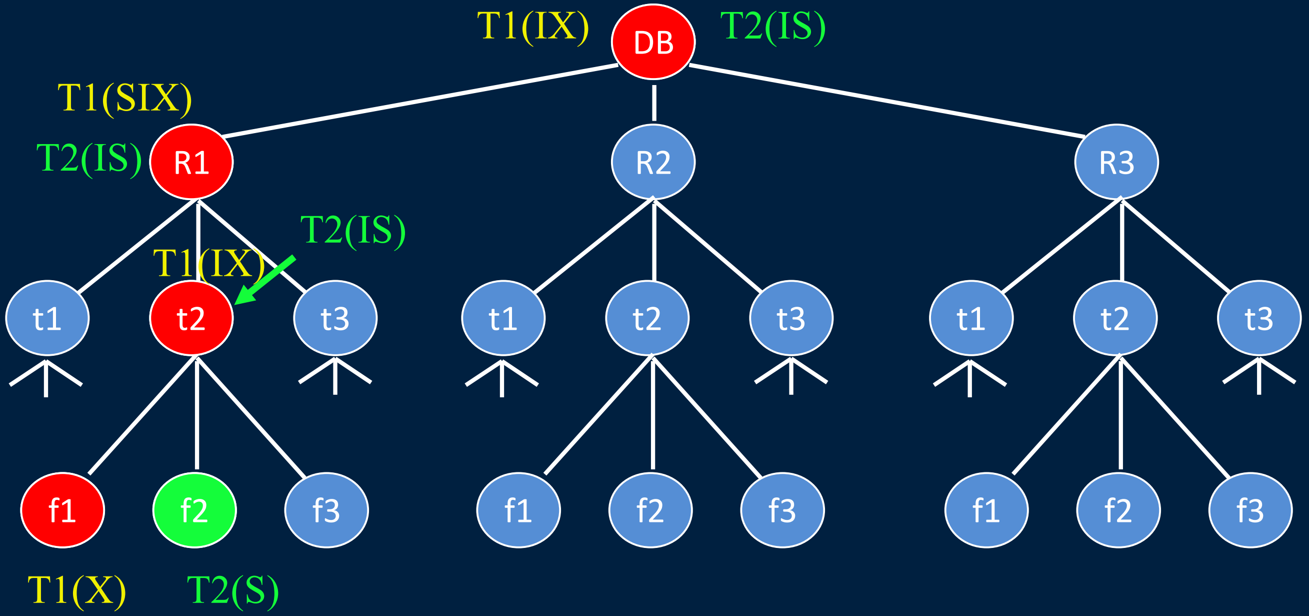
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (5)

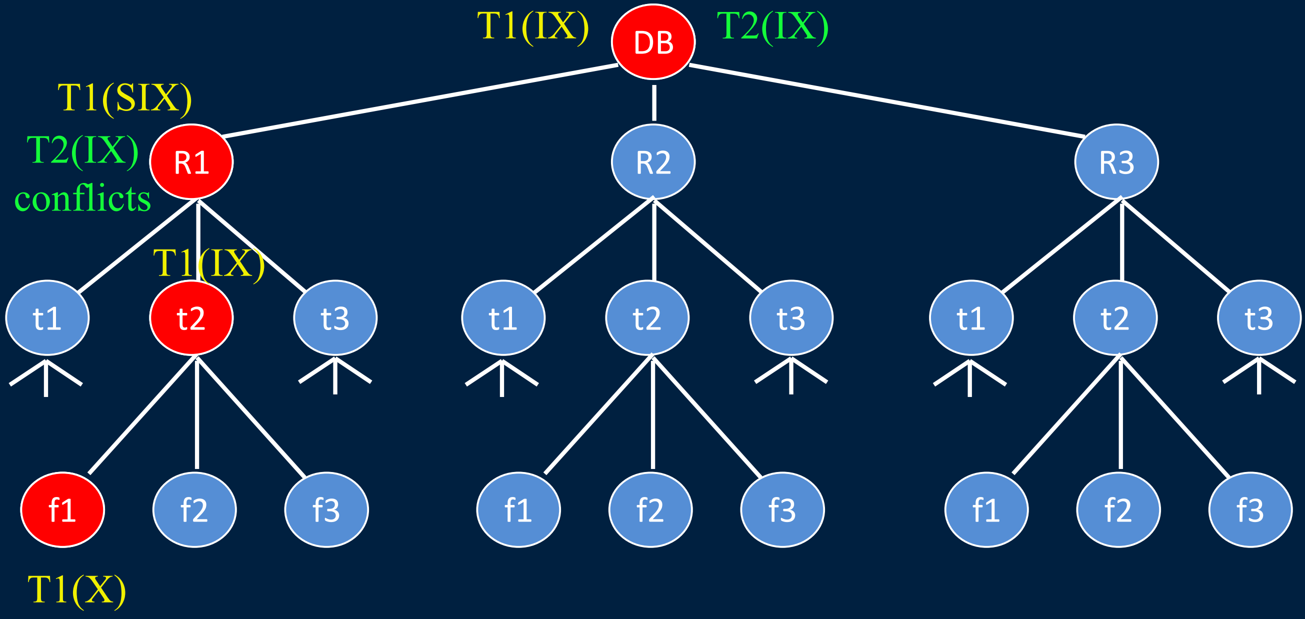
T2 wants to lock R1.t2.f2 in S-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (6)

T2 wants to lock R1.t2.f2 in X-mode. Does it work?



No, conflict at R1!

Multiple Granularity Locking Question

Question: How many of the following statements are true?

- i) The protocol always must lock the root node first.
- ii) If a child node is locked, its parent node must also be locked.
- iii) The protocol allows locking several tables at the same time.
- iv) The protocol is deadlock free.

A) 0

B) 1

C) 2

D) 3

E) 4

Deadlock Handling

A system is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Two mechanisms for deadlock handling:

- ***deadlock prevention*** - do not allow system to enter deadlock state
- ***deadlock detection*** - detect deadlock condition and abort transactions to remove deadlock state

Cost of deadlock handling includes:

- overhead of scheme itself
- potential losses in transaction processing due to rollbacks

Deadlock Prevention

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state.

Some strategies:

- Require that each transaction locks all its data items before it begins execution (predeclare locks, e.g. conservative 2PL).
- Impose a partial ordering on data items and require that a transaction lock data items only in the order specified.
- Wound-wait and wait-die strategies use timestamps to determine transaction age and determine if a transaction should wait or be rolled back on a lock conflict.

Wound-Wait and Wait-Die Strategies

Wait-Die scheme — non-preemptive

- Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
- A transaction may die several times before acquiring needed data item.

Wound-Wait scheme — preemptive

- Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
- May cause fewer rollbacks than *wait-die* scheme.

Note: A rolled back transaction is restarted with its original timestamp. Older transactions have precedence over newer ones, and starvation is avoided.

Timeout-Based Schemes

In a Timeout-Based Schemes:

- A transaction waits for a lock only for a specified amount of time. After that, the transaction times out and is rolled back.
- Thus deadlocks are not possible.
- Simple to implement, but starvation is possible.
- Difficult to determine good value of the timeout interval.
  - Too short - false deadlocks (unnecessary rollbacks)
  - Too long - wasted time while system is in deadlock

Deadlock Detection & Recovery

If deadlocks are not prevented, then a detection and recovery procedure is needed to recover when the system enters deadlock.

An algorithm is run periodically to check for deadlock. If the system is in deadlock, then transactions are aborted to resolve the deadlock.

Deadlock detection requires the system:

- Maintain information about currently allocated locks.
- Provide an algorithm to detect a deadlock state.
- Recover from deadlock by aborting transactions efficiently.

Wait-for Graphs

Deadlocks can be detected using a ***wait-for graph***, $G = (V,E)$:

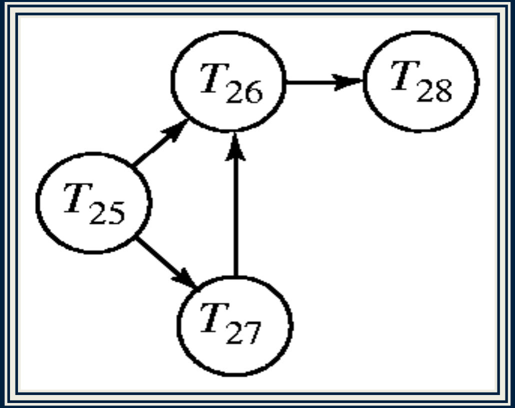
- V is a set of vertices (all the transactions in the system).
- E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted into the graph.

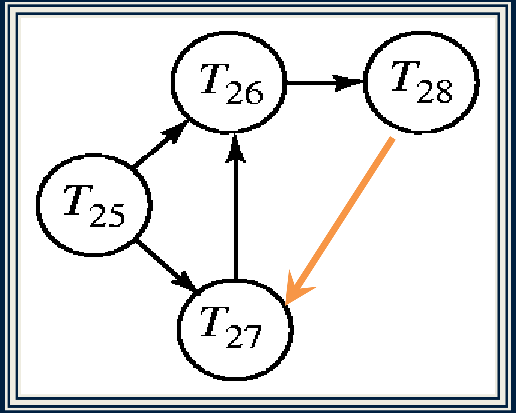
- This edge is removed only when T_j is no longer holding a data item needed by T_i.

The system is in a deadlock state if and only if the wait-for graph has a ***cycle***. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for Graph Examples



Wait-for graph with no cycle



Wait-for graph with a cycle

Deadlock Recovery

When a deadlock is detected three factors to consider:

- ***Victim selection*** - Some transaction will have to rolled back (made a victim) to break deadlock.
  - Select the victim transaction that will incur minimum cost (computation time, data items used, *etc.*).
- ***Rollback*** - determine how far to roll back transaction
  - **Total rollback:** Abort the transaction and then restart it.
  - More effective to roll back transaction only as far as necessary to break deadlock. (requires system store additional information)
- ***Starvation*** happens if same transaction is always chosen as victim.
  - Include the number of rollbacks in the cost factor to avoid starvation.

Deadlock Question

Question: How many of the following statements are true?

- i) A deadlock prevention protocol ensures deadlock never occurs.
- ii) In Wound-Wait, an older transaction waits on a younger one.
- iii) A wait-for graph has undirected edges between transactions.
- iv) A wait-for graph with 5 nodes but only 3 in a cycle is not in a deadlock state.

A) 0

B) 1

C) 2

D) 3

E) 4

Questions on Deadlocks

1) Assume a read-lock is requested before each read, and a write lock before each write. All unlocks occur after the last operation of a transaction. Explain what operations are denied during each schedule, draw the wait-for graph, and pick a transaction to abort if a deadlock does occur.

a) $r_1(A)$; r_2(B); w_1(C); r_3(D); $r_4(E)$; w_3(B); w_2(C); w_4(A); w_1(D);

b) $r_1(A)$; r_2(B); r_3(C); $w_1(B)$; w_2(C); w_3(D);

c) $r_1(A)$; $r_2(B)$; r_3(C); w_1(B); w_2(C); w_3(A);

Timestamp-Based Protocol

A ***timestamp protocol*** serializes transactions in the order they are assigned timestamps by the system.

Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.

- If an ***old*** transaction T_i has timestamp TS(T_i), a ***new*** transaction T_j has timestamp TS(T_j) where TS(T_i) < TS(T_j).
- The timestamp can be assigned using the system clock or some logical counter that is incremented for every timestamp.

Timestamp protocols do not use locks, so deadlock cannot occur!

Timestamp-Based Protocol
Read and Write Timestamps

To ensure serializability, the protocol maintains for each data *Q* two timestamp values:

- **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
- **R-timestamp**(*Q*) is the largest timestamp of any transaction that executed **read**(*Q*) successfully.

The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Timestamp-Based Protocol Rules

Suppose a transaction T_i issues a **read**(Q):

- If TS(T_i) < **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.
- If **TS(T_i)≥ W-timestamp(Q),** then the **read** operation is executed.
  - The R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Suppose that transaction T_i issues a **write**(Q):

- If **TS(T_i)≥ R-timestamp(Q) AND TS(T_i)≥ W-timestamp(Q),** then the **write** operation is executed.
- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was previously read by newer transaction.
  - Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. T_i is rolled back.

Timestamp Example

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5:

T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				
		write(Y)		
				read(Z)
	write(X) **abort**			
read(X)				
		write(Z) **abort**		
				write(Y) write(Z)

Correctness of Timestamp-Ordering Protocol

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph.

Timestamp protocol ensures freedom from deadlock as no transaction waits.

Protocol is not recoverable or cascade-free.

- Can achieve both properties if perform all writes atomically at end of the transaction.

Thomas' Write Rule

Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances:

- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Hence, rather than rolling back T_i as the timestamp ordering protocol would have done, this **write** operation can be ignored. Otherwise protocol is unchanged.

Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Timestamp Protocol Question

Question: How many of the following statements are true?

- i) Deadlock is not possible with timestamp protocols.
- ii) A transaction that arrives later to the system always has a smaller timestamp.
- iii) The precedence graph for the timestamp algorithm has edges from smaller timestamp transactions to larger ones.
- iv) A write is only performed if transaction has a timestamp >= the read timestamp for the data item.

A) 0

B) 1

C) 2

D) 3

E) 4

Questions on Timestamping

1) Indicate what happens during each of these schedules where concurrency control is performed using timestamps:

a) st_1; st_2; $r_1(A)$; r_2(B); w_2(A); $w_1(B)$;

b) st_1; $r_1(A)$; st_2; w_2(B); r_2(A); $w_1(B)$;

c) st_1; st_2; st_3; $r_1(A)$; $r_2(B)$; w_1(C); r_3(B); r_3(C); w_2(B); w_3(A);

d) st_1; st_3; st_2; $r_1(A)$; $r_2(B)$; w_1(C); r_3(B); r_3(C); w_2(B); w_3(A);

Validation Protocols

Validation or ***optimistic concurrency control protocols*** assume that the number of conflicts is low and verify correctness after a transaction is completed. Three phases:

- 1) ***Read phase*** – Transaction reads data items and performs operations. Writes are stored in local transaction memory.
- 2) ***Validation phase*** – Transaction checks if can proceed to write phase without violating serializability.
- 3) ***Write phase*** – All writes are copied to the database.

The validation test uses timestamps to guarantee that for two transactions T_i and T_j with $TS(T_i) < TS(T_j)$ either:

- 1) T_i finished before T_j started OR
- 2) Set of data items written by T_i does not intersect with items read by T_j and T_i completes writes before T_j validates.

Multiversion Schemes

Multiversion schemes keep old versions of data to increase concurrency. This is especially useful for read transactions.

Each successful **write** creates a new version of the data item. Use timestamps or transaction ids to label versions.

When a **read** operation is issued, select an appropriate version of the data item based on the timestamp.

Reads 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 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 Scheme

The following scheme ensures serializability:

- Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

If transaction T_i issues a **read**(Q) then:

- The value returned is the content of version Q_k.

If transaction T_i issues a **write**(Q):

- If TS(T_i) < R-timestamp(Q_k), then T_i is rolled back.
- If TS(T_i) = W-timestamp(Q_k), Q_k is overwritten.
- Otherwise a new version of Q is created.

Multiversion Timestamp Scheme (2)

Reads always succeed; writes may be 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.

Challenges:

- Must have an efficient way of handling versions (and discarding when no longer needed).
- Conflicts resolved through rollbacks rather than waiting so user application must be prepared to resubmit failed transactions.
  - Only update transactions can be rolled back.

Multiversion 2PL

Multiversion 2PL requires:

- 1) An integer counter used for timestamps for items and transactions.
- 2) Read-only transactions retrieve counter at start of transaction and use it to determine version to read. No locking used.
- 3) Update transactions perform rigorous 2PL. At commit, transaction increments timestamp counter and sets timestamp on every item it created.

Multiversion 2PL allows read transactions to never wait on locks and produces schedules that are recoverable and cascadeless.

Snapshot Isolation

Snapshot isolation is a widely-used protocol that gives each transaction its own "snapshot" of the database to execute on.

A snapshot consists of committed data values in the database before the transaction starts.

Read-only transactions never wait and are never aborted.

Update transactions keep updates private until commit when they are written to the database atomically. A validation is performed before writing the updates are allowed.

Snapshot Isolation Validation Test

Two ways to validate:

First committer wins:

- Transaction *T* enters prepared to commit state and checks:
  - If any concurrent transaction has updated any item *T* wants to update.
  - If yes, *T* is aborted. If no, *T* commits and updates written to database.

First update wins:

- If transaction T wants to update, it must get write lock on item.
- When lock is acquired, check if item has been updated by a concurrent transaction. If so, abort, otherwise proceed.

Snapshot Isolation Serializability Issues

Despite its advantages and being widely implemented (Oracle, PostgreSQL, SQL Server), snapshot isolation does not ensure serializability.

There are cases where particular transaction schedules are not serializable.

However, these issues can be often ignored or avoided, especially since primary and foreign key constraints are validated after snapshot validation and will often detect conflicts.

Multiversion and Snapshot Isolation Question

Question: How many of the following statements are true?

- i) Reads always succeed with a multiversion scheme.
- ii) Writes always succeed and create a new version each write.
- iii) Snapshot isolation guarantees serializability.
- iv) In a multiversion scheme, a read for a transaction may occur on a data value that is not the most recent.

A) 0

B) 1

C) 2

D) 3

E) 4

Insert and Delete Operations

In addition to read/write operations, the system must handle ***delete*** and ***insert*** operations.

Deletion with two-phase locking:

- May only be performed if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.

Insertion with two-phase locking:

- A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.

The Phantom Phenomenon

Inserts/deletes can lead to the ***phantom phenomenon***:

- A transaction that scans a relation (e.g., find all students) and a transaction that inserts a tuple in the relation (e.g., inserts a new student) may conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result: the scan transaction may not see the new tuple, yet may be serialized after the insert transaction.
- Transactions conflict over a ***phantom tuple***.

The transaction scanning the relation reads information that indicates what tuples the relation contains. A transaction inserting a tuple updates the same info.

This information should be locked.

The Phantom Phenomenon (2)

Can prevent problem by:

- Accepting the issue (read committed isolation)
- Locking the entire relation (multi-granularity locking)
- Using index-locking or predicate-locking to guarantee that conflicts within the relation are detected.
- Having a special lock associated with the entire file. Read transactions that scan the whole relation must get a read lock on it and update transactions must get a write lock.

Transaction Definition in SQL

In SQL, a transaction begins implicitly.

A transaction in SQL ends by:

- **Commit** accepts updates of current transaction.
- **Rollback** aborts current transaction and discards its updates. Failures may also cause a transaction to be aborted.

An ***isolation level*** reflects how a transaction perceives the results of other transactions. It applies only to your perspective of the database, not other transactions/users. Lowering isolation level improves performance but may potentially sacrifice consistency.

Example Transactions

Transaction to deposit $50 into a bank account:

```
BEGIN TRANSACTION;
   UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT; -- T1
```

Transaction to calculate totals for all accounts (twice):

```
BEGIN TRANSACTION;
   SELECT SUM(balance) as total1 FROM Account;
    SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

Transaction to add a new account:

```
BEGIN TRANSACTION;
   INSERT INTO ACCOUNT (num, balance) VALUES ('S5', 100);
COMMIT T3;
```

Levels of Consistency in SQL-92

The isolation level can be specified by:

`SET TRANSACTION ISOLATION LEVEL =` *X* where *X* is

- **Serializable** - transactions behave like executed one at a time.
- **Repeatable read** - repeated reads must return same data. Does not necessarily read newly inserted records.
- **Read committed** - only committed values can be read, but successive reads may return different values.
- **Read uncommitted** - even uncommitted records may be read. Reading an uncommitted value is called a *dirty read*.

Scheduling of Transactions

Each transaction in a database is a separate executing program.

- A transaction may be its own program or a thread of execution.

The operating system schedules the execution of programs outside of the control of the DBMS.

- Thus, transactions may be executed in any order (as long as the order of operations within a transaction are the same). This interleaving is what produces different schedules.

The DBMS uses its concurrency control protocol to restrict the schedules to those that respect the consistency specified by the user for the transaction isolation level.

- All transactions must write lock any data item updated and the relation lock if inserting.
- Isolation level only affects read locks.

Scheduling Question

Question: **TRUE** or **FALSE**: The database has complete control over the scheduling of transactions.

A) True

B) False

Isolation Example
Serializable

A ***serializable*** schedule requires that regardless of the interleaving of the operations, the final result is the same as some serial ordering of the transactions.

- Read and write locks are held to commit. Also have a relation-level lock.

For three transactions, there are 3! = 6 serial schedules.

For these examples, assume that the total amount of money in all accounts is $5000 before the transactions begin.

Isolation Example Serializable (2)

Example schedule for T1, T2, T3:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
SELECT SUM(balance) as total1 FROM Account;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
```

After execution, total1 = $5050 and total2 = $5050.

- The results for all six serial schedules are:
  - T1, T2, T3 – total1 = $5050 ; total2 = $5050
  - T1, T3, T2 – total1 = $5150 ; total2 = $5150
  - T2, T1, T3 – total1 = $5000 ; total2 = $5000
  - T2, T3, T1 – total1 = $5000 ; total2 = $5000
  - T3, T1, T2 – total1 = $5150 ; total2 = $5150
  - T3, T2, T1 – total1 = $5100 ; total2 = $5100

Isolation Example
Repeatable read

With ***repeatable read***, a transaction is guaranteed to get the same data back on multiple reads but may see ***phantom records*** inserted in between reads.

- Read and write locks are held to commit.

Example schedule:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
SELECT SUM(balance) as total1 FROM Account;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

After execution, total1 = $5050 and total2 = $5150 as the second read sees the newly inserted tuple.

Isolation Example
Read Committed

With ***read committed***, each read will get the most recently committed values even if different than an earlier read.

- Read locks are released after every statement. Write locks released at commit.

Example schedule:

```
SELECT SUM(balance) as total1 FROM Account;
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

After execution, total1 = $5000 and total2 = $5150 as the second read sees the newly inserted tuple and T1’s update.

Isolation Example
Read Uncommitted

Read uncommitted allows a transaction to read dirty data that has not been (and may never be) committed.

- Transaction acquires no read locks.

Example schedule:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
SELECT SUM(balance) as total1 FROM Account;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
ABORT T3;
ABORT T1;
```

After execution, total1 = $5050 and total2 = $5150 as T2's sees even uncommitted data. Note that both T1 and T3 abort so T2 sees incorrect data. **It is very dangerous to use read uncommitted if the transaction updates the database!**

Summary of Isolation Levels

Isolation Level	Problems	Lock Usage	Speed	Comments
Serializable	None	Read locks held to commit ; read lock on relation	Slowest	Only level that guarantees correctness.
Repeatable read	Phantom tuples	Read locks held to commit	Medium	Useful for modify transactions.
Read committed	Phantom tuples, values may change	Read locks released after each statement	Fast	Useful for transactions where operations are separable but updates are all or none.
Read uncommitted	Phantoms, values may change, dirty reads	No read locks	Fastest	Useful for read-only transactions that tolerate inaccurate results

Isolation Levels Question

Question: How many of the following statements are true?

- i) Serializability guarantees that there are no phantom tuples.
- ii) Read committed may be affected by phantom tuples.
- iii) In read committed, two reads at separate times may retrieve different values.
- iv) Read uncommitted is the fastest isolation level.

A) 0

B) 1

C) 2

D) 3

E) 4

Transaction Practice Question

Given these transactions and table `Bid(itemID, price)` that initially contains the two tuples: `(i1,10)` and `(i2,20)`:

```
T1:  BEGIN TRANSACTION;
          S1:  UPDATE Bid SET price = price + 5;
          S2:  INSERT INTO Bid VALUES (i3,30);
          COMMIT;

T2:  BEGIN TRANSACTION;
          S1:  SELECT SUM(price) AS p1 FROM Bid;
          S2:  SELECT MAX(price) AS p2 FROM Bid;
          COMMIT;
```

Assume that T1 executes with isolation level serializable and both transactions successfully commit.

- 1) If T2 executes with isolation level serializable, what are all the possible pairs of values for p1 and p2 returned by T2?
- 2) If T2 executes with isolation level read committed, what are all the possible pairs of values for p1 and p2 for T2?

Concurrency Control in PostgreSQL

PostgreSQL uses snapshot isolation for DML and 2PL for DDL.

- Snapshot isolation implementation is referred to as multi-version concurrency control (MVCC).
  - Uses first updater wins policy. Uses x-locks on written rows.
  - Each transaction has id (logical counter). Each tuple has transaction id that created it. Keeps track of snapshot info for each transaction.
  - Tradeoff: Reads never wait but more space used that must be handled.
- Uses deadlock detection with timeouts (default 1 sec.).

Isolation levels supported:

- read committed (default), serializable
  - For read committed, timestamp is at statement level. For serializable, timestamp is transaction's first timestamp.
  - A transaction will wait for a lock on a row currently being updated. If update committed by another transaction, waiting transaction issues error "could not serialize access due to concurrent update". Only possible for update/deletes.

Concurrency Control in MySQL

MySQL with the InnoDB storage engine uses snapshot isolation (multi-version concurrency control) for reads and 2PL for updates.

Supports all 4 isolation levels with different locks acquired for different levels. Default is repeatable read.

Concurrency Control in Microsoft SQL Server

Microsoft SQL Server uses 2PL and optimistic concurrency control.

Supports all four isolation levels plus two snapshot isolation levels.

Uses multiple granularity locking and automatically determines correct sizes (table, extent, page, rows).

Older snapshots are stored in temporary database.

Deadlock detection performed every 5 seconds by default.

Concurrency Control in Oracle

Oracle uses ***multiversion read consistency*** (snapshots).

- No locks for a read operation, so a read never blocks for a write.
- Uses row-level locking and transaction will wait if tries to change row updated by uncommitted transactions.
- System change number (SCN) used for ordering operations.
- Stores row lock on data block where row is stored.
- Locks held throughout transaction, released at commit/abort
  - Different types of locks; DDL, DML, mutex, latches
- Does deadlock detection using wait-for graphs
- Oracle Flashback Technology allows recovering a table to a point in time. Can be used to recover deleted rows or dropped tables without doing full restore from backup.

Implements: read committed and serializable isolation levels

Concurrency Control in MongoDB

MongoDB is a NoSQL document database. Performs atomic updates at document-level with no support for transactions.

MongoDB does not support any of the traditional isolation levels directly.

Uses reader-writer locks to ensure a data item can be read by many but only written by one at a time.

- Waiting writers have precedence over readers.
- Until Mongo 3.0, locking was at the database level. Mongo 3.0 and above perform multiple granularity locking (database, collection, document).

Concurrency Control Summary

Concurrency control protocols are used to ensure concurrent transactions maintain their isolation.

- ***Two-phase locking*** (2PL) and multigranularity locking schemes are commonly used.
- ***Deadlocks*** must be handled by either deadlock prevention or deadlock detection and recovery.
  - Prevention: wound-wait and wait-die schemes
  - Detection: wait-for graphs and transaction rollback

Multiversion schemes and snapshots create new versions on every update and determine the correct version for reads.

- Allows higher concurrency but uses more space. Very common.

SQL isolation levels are read uncommitted, read committed, repeatable read, and serializable.

- Differ on handling of dirty reads and phantom tuples.

Major Objectives

The "One Things":

- Explain how two-phase locking (2PL) works and detect valid 2PL schedules.
- Perform deadlock detection and recovery using wait-for graphs.
- Explain and use the timestamp based protocol.
- Perform multiple granularity locking using lock modes, rules, and compatibility matrix.
- Understand difference between snapshot based approaches (MVCC) and using 2PL.

Objectives

- Define concurrency control, locking protocol, deadlock, starvation, exclusive and shared locks (compatibility matrix).
- Define and use conservative, strict, and rigorous 2PL.
- Explain the use of lock conversions (upgrades/downgrades).
- Insert locks into a schedule using automatic algorithm.
- List some methods for deadlock prevention.
- List three factors with deadlock recovery.
- Define and motivate a validation based protocol.
- Explain the motivation for multiversion 2PL and timestamping.
- Explain the general approach for snapshot protocols.
- Explain how the phantom phenomenon occurs.
- List consistency levels in SQL-92 and determine which schedules are valid under each consistency level.


UBC
THE UNIVERSITY OF BRITISH COLUMBIA