Chapter 10 – Concurrency Control Techniques

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. Purpose of Concurrency Control is to enforce Isolation (through mutual exclusion) among conflicting transactions there by preserve database consistency.

Techniques for Concurrency Control

1. Locking data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols have been proposed.
2. Use of timestamps. A timestamp is a unique identifier for each transaction, generated by the system.
3. Multiversion concurrency control protocols that use multiple versions of a data item.
4. Optimistic Concurrency Control: based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols.**

Locking Techniques for Concurrency Control

A **lock is** a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database.

Types of Locks:

- **Binary locks:** only two states of a lock; too simple and too restrictive; not used in ractice.
- **Shared/exclusive locks:** which provide more general locking capabilities and are used in practical database locking schemes. (Read Lock as a shared lock, Write Lock as an exclusive lock).
- **Certify lock:** used to improve performance of locking protocols.
- **Intension Locks:** used for locking data items of multiple granularities.

Binary locks

- A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity).
- A binary lock enforces **mutual exclusion** on the data item; i.e., at a time only one transaction can hold a lock.
- A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested.
- **LOCK(X)** : indicates the current value (or state) of the lock associated with item X

Binary Locking Scheme

Every transaction must obey the following rules. Rules are enforced by the LOCK MANAGER

- A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
- A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
- A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
- A transaction T will not issue an unlock_item(X) operation on X unless it already holds the lock on item X.

Shared/Exclusive (or Read/Write) locks

- A lock associated with an item X, LOCK(X), now has three possible states: "Read-locked", "Write-locked" or "Unlocked."
- A read-locked item is also called share-locked, because other transactions are allowed to read the item.
- A write-locked item is called exclusive-locked, because a single transaction exclusively holds the lock on the item.

Rules for Read/Write Locks

- A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
- A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
- A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
- A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
  (EXCEPTIONS: DOWNGRADING OF LOCK from WRITE TO READ)
- A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
  (EXCEPTIONS: UPGRADING OF LOCK FROM READ TO WRITE)
- A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

UPGRADING:

If T *is the only transaction* holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait.

DOWNGRADING:

It is also possible for a transaction T to issue a write_lock(X) and then later on to downgrade the lock by issuing a read_lock(X) operation.

2 PHASE LOCKING PROTOCOL

Two-phase locking (**2PL**) is a concurrency control method that guarantees serializability.

Guaranteeing Serializability by Two-phase Locking

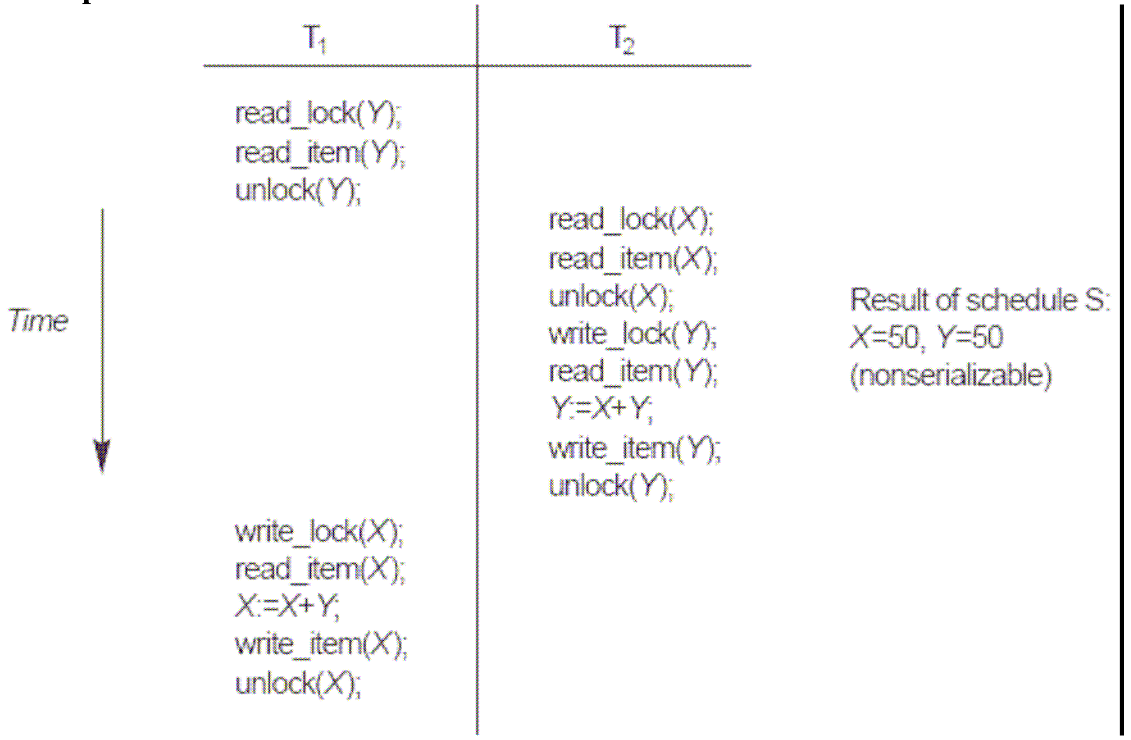
By the 2PL protocol, locks are applied and removed in two phases:

1. **Expanding phase:** locks are acquired and no locks are released.
2. **Shrinking phase:** locks are released and no locks are acquired.

WITH LOCK CONVERSION:

Upgrading of locks (from read-locked to write-locked) must be done during the expanding phase,
Downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

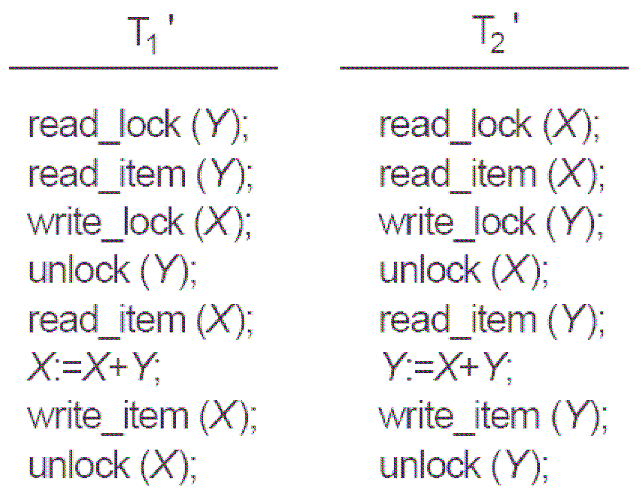
Example:



T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

Time
Result of schedule S:
X=50, Y=50
(nonserializable)

Figure1. Transactions that do not obey two- phase locking.



T_1 '	T_2 '
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

Figure2. Transactions that obey two- phase locking but they can produce deadlock.

Claims:

The 2PL protocol guarantees serializability

- Any schedule of transactions that follow 2PL will be serializable
- We therefore do not need to test a schedule for serializability

But, it may limit the amount of concurrency since transactions may have to hold onto locks longer than needed, creating the new problem of deadlocks.

Variations of 2-Phase Locking

- **Conservative 2PL (Or Static 2PL):** Requires a transaction to lock all the items it accesses *before the transaction begins execution*, by predeclaring it's *read-set* and *write-set*.(the read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that it writes.) If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.

  POLICY - Lock all that you need before reading or writing. Transaction is in shrinking phase after it starts.

  PROPERTY: Conservative 2PL is a deadlock-free protocol

  PRACTICAL: Difficult to use because of difficulty in predeclaring the read-set and write-set.

- **Strict 2PL:** The most popular variation of 2PL is strict 2PL, which guarantees strict schedules. In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.

POLICY - Release write locks only after terminating. Transaction is in expanding phase until it ends (may release some read locks before commit).
PROPERTY: NOT a deadlock-free protocol

PRACTICAL : Possible to enforce and desirable due to recoverability.

- **Rigorous 2PL:** A transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts Behaves similar to Strict 2 PL except it is more restrictive, but easier to implement since all locks are held till commit.

Deadlock

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item X, but X is locked by another transaction T ' in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.

Deadlock Prevention

1. **Use Conservative Locking**: every transaction *lock all the items it needs in advance* (generally not a practical assumption) —if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs.
2. **Ordering all the items** in the database and making sure that a transaction that needs several items will lock them according to that order.
3. **Use of transaction timestamp** TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically ordered based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2).
   Notice that the *older* transaction T1 has the *smaller* timestamp value.
   Two schemes that prevent deadlock based on time stamps are wait-die and wound-wait. Suppose that transaction Ti tries to lock an item X, but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rules followed by these schemes are as follows:
   - **wait-die**: if TS(Ti) < TS (Tj) then (Ti older than Tj) Ti is allowed to wait otherwise (Ti younger than Tj) abort Ti (Ti *dies*) and restart it later *with the same timestamp.*
   - **wound-wait**: if TS(Ti) < TS(Tj) then (Ti older than Tj) abort Tj (Ti *wounds* Tj) and restart it later *with the same timestamp.* otherwise (Ti younger than Tj) Ti is allowed to wait
4. **No waiting**: In case of inability to obtain a lock, a transaction aborts and is resubmitted with a fixed delay. (Causes too many needless aborts).
5. **Cautious Waiting**: Suppose that transaction Ti tries to lock an item X but is not able to do so because X is locked by some other transaction Tj with a conflicting lock.if Tj is not blocked (not waiting for some other locked item) then Ti is blocked and allowed to wait otherwise abort Ti i.e., if X is waiting for Y, let it wait unless Y is also waitiing for Z to release some other item.

Starvation

1. A transaction is starved if it cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others.
2. Starvation can also occur in the algorithms for dealing with deadlock. Occurs if the algorithms select the same transaction as victim repeatedly, thus causing it to abort and never finish execution.

Remedies for Preventing Starvation

1. First-come-first-serve queue - a fair waiting scheme; - transactions are enabled to lock an item in the order in which they originally requested to lock the item.
2. Allow some transactions to have priority over others but increase the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
3. The victim selection algorithm can use higher priorities for transactions that have been aborted multiple times so that they are not selected as victims repeatedly.

The wait-die and wound-wait schemes avoid starvation.

Concurrency Control based on Timestamp Ordering

- **Timestamp:** a unique identifier created by the DBMS to identify a transaction. A timestamp can be thought of as the *transaction start time*.
- Timestamp of transaction T is referred to as **TS(T)**.
- **NO deadlocks** occur because there are no locks.

Timestamp Ordering Algorithm:

The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. **read_TS(X)**: The read timestamp of item X; this is the largest timestamp among all the timestamps of transactions that have successfully read item X (that is, read_TS(X) = TS(T), where T is the *youngest* (latest) transaction that has read X successfully).
2. **write_TS(X)**: The write timestamp of item X; this is the largest of all the timestamps of transactions that have successfully written item X (that is, write_TS(X) = TS(T), where T is the *youngest* (latest) transaction that has written X successfully).

Basic TO Algorithm

- Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, The **basic TO** algorithm compares the timestamp of T with the read timestamp: read_TS(X) and the write timestamp: write_TS(X) to ensure that the timestamp order of transaction execution is not violated.

- If the timestamp order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as **cascading rollback - a problem with the basic TO algorithm.**

Algorithm:

1. Transaction T issues a write_item(X) operation:
   (a) If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation.
   (b) If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).
2. Transaction T issues a read_item(X) operation:
   (a) If write_TS(X) > TS(T), then abort and roll back T and reject the operation.
   (b) If write_TS(X) <= TS(T), then execute the read_item(X) operation of T and set read_TS(X) to TS(T).

Variations of Timestamp Ordering

- **Strict Timestamp Ordering :** A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.
  - A transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T' that *wrote* the value of X (hence TS(T') = write_TS(X)) has committed or aborted.

- **Thomas's Write Rule:** **Thomas' write rule,** does not enforce conflict serializability; but it rejects fewer write operations, by *modifying the checks for the* ***write_item(X)*** *operation as follows*:
  1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
  2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. [THIS IS A REJECTED WRITE. This is because some transaction with timestamp greater than TS(T) and hence after T in the timestamp ordering has already written the value of X. Hence, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (a).]
  3. If neither the condition in part (a) nor the condition in part (b) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

Multiversion Concurrency Control Techniques

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect: Significantly more storage (RAM and disk) is required to maintain multiple versions.

Multiversion technique based on timestamp ordering

Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.

- **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
- **write_TS(Xi)**: The write timestamp of Xi is the timestamp of transaction that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.
- To ensure serializability, the following two rules are used.
  - If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and set read_TS(X) = write_TS(Xj) = TS(T).
  - If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).
- Rule 2 guarantees that a read will never be rejected.

Multiversion Concurrency control technique using Certify Locks

- **Concept:** Allow a transaction Ti to read a data item X while it is write locked by a conflicting transaction Tj. This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction.
- **Steps**
  - X is the committed version of a data item.
  - T creates a second version X' after obtaining a write lock on X.
  - Other transactions continue to read X.
  - T is ready to commit so it obtains a certify lock on X'.
  - The committed version X becomes X'.
  - T releases it's certify lock on X', which is X now.

Compatibility matrix for certify locks:-

	Read	Write	Certify
Read	yes	yes	no
Write	yes	no	no
Certify	no	no	no

Validation (Optimistic) Concurrency Control Scheme

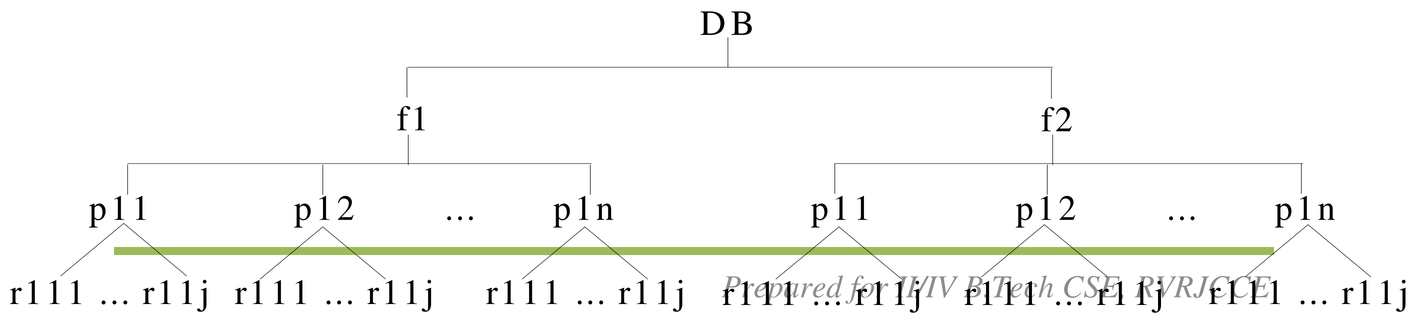
- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.
- Using this technique, transactions are executed in three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**
- **Read phase**: A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).
- **Validation phase**: Serializability is checked before transactions write their updates to the database. This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
  1. Tj completes its write phase before Ti starts its read phase.
  2. Tj completes its write phase before Ti starts its write phase, and the read_set of Ti has no items in common with the write_set of Tj
  3. Tj completes its read phase before Ti completes its read phase and, both the read_set and write_set of Ti have no items in common with the write_set of Tj.

  When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and Ti is aborted.
- **Write phase**: On a successful validation transactions updates are applied to the database; otherwise, transactions are restarted.

Granularity of data items and Multiple Granularity Locking

- A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.

The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

- **Intention-shared (IS)**: indicates that a shared lock(s) will be requested on some descendent nodes(s).
- **Intention-exclusive (IX)**: indicates that an exclusive lock(s) will be requested on some descendent node(s).
- **Shared-intention-exclusive (SIX)**: indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

The **multiple Granularity 2PL(2PhaseLocking) locking protocol** is an appropriate protocol that makes use of these locks for ensuring serializability. It consists of the following rules:

1. The lock compatibility must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N, only if none of the children of N are currently locked by T.