Concurrency Control Techniques

Database Concurrency Control

When several transactions execute concurrently in the database, the database system must control the interaction among the concurrent transactions.

Purpose of Concurrency Control

- To enforce Isolation among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Database Concurrency Control

Two-Phase Locking(2PL) Techniques

Locking is an operation which secures (a) permission to Read or (b) permission to Write a data item for a transaction. Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: Unlock (X). Data item X is made available to all other transactions.

Lock and Unlock are Atomic operations.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Two locks modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared or exclusive lock can be applied by any other transaction on X.

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Database requires that all transactions should be well-formed. A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
   then LOCK (X) ← 1 (*lock the item*)
   else begin
            wait (until lock (X) = 0) and
              the lock manager wakes up the transaction);
            goto B
            end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
    wake up one of the waiting the transactions;

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the read lock operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
            no_of_reads (X) ← 1;
        end
        else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
             else begin wait (until LOCK (X) = "unlocked" and
                        the lock manager wakes up the transaction);
                        go to B
                     end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
        LOCK (X) ← 1 "write-locked";
        else begin wait (until LOCK (X) = "unlocked" and
                        the lock manager wakes up the transaction);
                        go to B
                    end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
        begin LOCK (X) ← "unlocked";
              wakes up one of the transactions, if any
        end
        else if LOCK (X) ← "read-locked" then
             begin
                  no_of_reads (X) ← no_of_reads (X) -1
                  if  no_of_reads (X) = 0 then
                  begin
                         LOCK (X) = "unlocked";
                        wake up one of the transactions, if any
                  end
             end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock conversion

Lock upgrade: existing read lock to write lock

if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then
    convert read-lock (X) to write-lock (X)
else
    force Ti to wait until Tj unlocks X

Lock downgrade: existing write lock to read lock

Ti has a write-lock (X) (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two Phases: (a) Locking (Growing) (b) Unlocking (Shrinking).

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Both these transactions do not follow the two phase locking protocol.

T1	T2
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
unlock (Y);	unlock (X);
write_lock (X);	Write_lock (Y);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2
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 ↓

This schedule is Nonserializable because it. violated two-phase policy.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1	T'2	
read_lock (Y);	read_lock (X);	T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.
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);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.

Basic: Transaction locks data items **incrementally**. This may cause deadlock which is dealt with.

Conservative or static: Prevents deadlock by locking all desired data items before transaction begins execution.

Strict: A more stricter version of Basic algorithm where unlocking of all exclusive locks is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Rigorous: Unlocking of all its locks (exclusive or shared) is performed after a transaction terminates (commits or aborts and rolled-back). Hence, it is easier to implement than strict version.

Lock Conversion: We can also refine the basic two-phase locking protocol in which lock conversions are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock , and downgrading an exclusive lock to a shared lock to get more concurrency.

Lock conversion cannot be allowed arbitrarily. Upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

T1	T2
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlock

Deadlock (T1 and T2)

Database Concurrency Control

Dealing with Deadlock and Starvation

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.

Deadlock prevention

A transaction locks all data items it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. The conservative two-phase locking uses this approach.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock avoidance

There are many variations of two-phase locking algorithm. Some avoid deadlock by not letting the cycle to complete. That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction. Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Wait-die: 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: Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

Database Concurrency Control

Dealing with Deadlock and Starvation

Starvation

Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back. This limitation is inherent in all priority based scheduling mechanisms. In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Database Concurrency Control

Timestamp based concurrency control algorithm

Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Database Concurrency Control

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
   b. If the condition in part (a) does not exist, then execute write_item(X) 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 an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
   b. If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Database Concurrency Control

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction T issues a write_item(X) operation:
   a. If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
   a. If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Database Concurrency Control

Timestamp based concurrency control algorithm

Thomas's Write Rule

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 just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

Database Concurrency Control

Multiversion concurrency control techniques

Concept

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. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Database Concurrency Control

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 that wrote the value of version Xi.

A new version of Xi is created only by a write operation.

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

1. 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 Xj of X with read_TS(Xj) = write_TS(Xj) = TS(T).

2. 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.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Concept

There are *three locking modes* for an item: **read**, **write**, and **certify**, instead of just the two modes discussed previously.

Allow a transaction T’ to read a data item X while it is write locked by a conflicting transaction T.

This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. The second version X’ is created when a transaction T acquires a write lock on the item.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Steps

1. T creates a second version X' after obtaining a write lock on committed version X.
2. Other transactions can continue to read X while T holds the write lock on it.
3. Transaction T can write the value of X' as needed, without affecting the value of the committed version X.
4. Once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit.
5. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks.
6. Once the certify locks are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

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

read/write/certify locking scheme

Note: In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently. This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort, since transactions are only allowed to read the version X that was written by the committed transaction. but may get deadlocked if upgrading of a read lock to a write lock is allowed.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

Three phases:

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: Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.

This phase requires *transaction timestamps*, *write_sets* (set of items the transaction writes), *read_sets* (set of items the transaction reads); in addition, *start* and end *times* from some of the three phases.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

The validation 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. Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj
3. Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase before Ti completes its read phase.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

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: If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

Database Concurrency Control

Concurrency Control based on Snapshot Isolation

In this scheme, read operations do not require read locks to be applied to the items, thus reducing the overhead associated with two-phase locking. However, write operations do require write locks. Thus, for transactions that have many reads, the performance is much better than 2PL. When writes do occur, the system will have to keep track of older versions of the updated items in a temporary version store (sometimes known as tempstore), with the timestamps of when the version was created. This is necessary so that a transaction that started before the item was written can still read the value (version) of the item that was in the database snapshot when the transaction started.

To keep track of versions, items that have been updated will have pointers to a list of recent versions of the item in the tempstore, so that the correct item can be read for each transaction. The tempstore items will be removed when no longer needed, so a method to decide when to remove unneeded versions will be needed.