Jisy Raju
Assistant Professor, CE Cherthala

Module 5

Transactional concurrency control:- Transactions, Nested Transactions-Locks-Optimistic concurrency control

5.1 TRANSACTIONS

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

A's Account
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)

B's Account
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)

In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called **ACID** properties.

- **Atomicity:**
  By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations. Atomicity is also known as the 'All or nothing rule'.
  —Abort: If a transaction aborts, changes made to database are not visible.
  —Commit: If a transaction commits, changes made are visible.

  Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2.(say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

- **Consistency:**
  This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to correctness of a database. Referring to the example above,
  The total amount before and after the transaction must be maintained.
  Total before T occurs = 500 + 200 = 700.
  Total after T occurs = 400 + 300 = 700.
  Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

- **Isolation:**
  This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

- **Durability:**
  This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even is system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost

Transaction sequence must continue until:

- COMMIT statement is reached
- ROLLBACK statement is reached
- End of program is reached
- Program is abnormally terminated

SQL statements that provide transaction support,

- COMMIT
- ROLLBACK

Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator.

Operations in the *Coordinator* interface

openTransaction() → trans;
Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans)→ (commit, abort);
Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
Aborts the transaction.

Service actions related to process crashes • If a server process crashes unexpectedly, it is eventually

replaced. The new server process aborts any uncommitted transactions and uses a recovery procedure to restore the values of the objects to the values produced by the most recently committed transaction. To deal with a client that crashes unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

Client actions related to server process crashes • If a server crashes while a transaction is in progress, the client will become aware of this when one of the operations returns an exception after a timeout. If a server crashes and is then replaced during the progress of a transaction, the transaction will no longer be valid and the client must be informed via an exception to the next operation. In either case, the client must then formulate a plan, possibly in consultation with the human user, for the completion or abandonment of the task of which the transaction was a part.

5.2 Concurrency control

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules. **Serializability** is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.The various approaches for concurrency control,

- Lock based concurrency control
- Timestamp concurrency control
- Optimistic concurrency control

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.

- **Lost Update:**
- **Dirty Read Problems**
- **Inconsistent Retrievals**

Lost Update: This problem occurs when two transactions that access the same database items, have their operations in a way that makes the value of some database item incorrect.

In other words, if transactions T1 and T2 both read a record and then update it, the effects of the first update will be overwritten by the second update.

Example:

Consider the situation given in figure that shows operations performed by two transaactions, Transaction-A and Transaction- B with respect to time.

Transaction- A	Time	Transaction- B
-----	t0	----
Read X	t1	----
----	t2	Read X
Update X	t3	----
----	t4	Update X
----	t5	----

At time t1 , Transactions-A reads value of X.

At time t2 , Transactions-B reads value of X.

At time t3,Transactions-A writes value of X on the basis of the value seen at time t1.

At time t4,Transactions-B writes value of X on the basis of the value seen at time t2.

So,update of Transactions-A is lost at time t4,because Transactions-B overwrites it without looking at its current value.

Such type of problem is referred as the Update Lost Problem, as update made by one transaction is lost here

Dirty Read Problems: This problem occurs when one transaction reads changes the value while the other reads the value before committing or rolling back by the first transaction.

Example:

Consider the situation given in figure :

Transaction- A	Time	Transaction- B
----	t0	----
----	t1	Update X
Read X	t2	----
----	t3	Rollback
----	t4	----

At time t1 , Transactions-B writes value of X.

At time t2 , Transactions-A reads value of X.

At time t3 , Transactions-B rollbacks.So,it changes the value of X back to that of prior to t1.

So,Transaction-A now has value which has never become part of the stable database.

Such type of problem is referred to as the Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

Inconsistent Retrievals: occurs when a transaction calculates some summary (aggregate) function over a

set of data while other transactions are updating the data.

example: let x=10, y=20, z=20

T1	T2
R(x)	
R(y)	
sum=x+y	
	R(z)
	w(z)
	z=z-10
R(z)	commit
sum=sum+z	

Serializability: Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database(Interleaved execution of transactions yields the same results as the serial execution of the transactions.)

Conflicting operations: Two operations are called as conflicting operations if all the following conditions hold true for them-

- Both the operations belong to different transactions
- Both the operations are on same data item
- Read – Read (No conflict)
- Read – Write (or Write – Read)Conflict!
- Write – Write (Conflict)

Example-

Consider the following schedule-

Transaction T1	Transaction T2
R1 (A)	
W1 (A)	
	R2 (A)
R1 (B)	

In this schedule, W1 (A) and R2 (A) are called as conflicting operations because all the above conditions hold true for them.

5.2.1 Recoverability from aborts

Servers must record all the effects of committed transactions and none of the effects of aborted transactions.

This section illustrates two problems associated with aborting transactions,

- dirty reads
- premature

Dirty reads • The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions. The 'dirty read' problem is caused by the interaction between a read operation in one transaction and an earlier write operation in another transaction on the same object

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

T2 sees result update by T1 on account A

T2 performs its own update on A & then commits.

T1 aborts -> T2 has seen a "transient" value

T2 is not recoverable

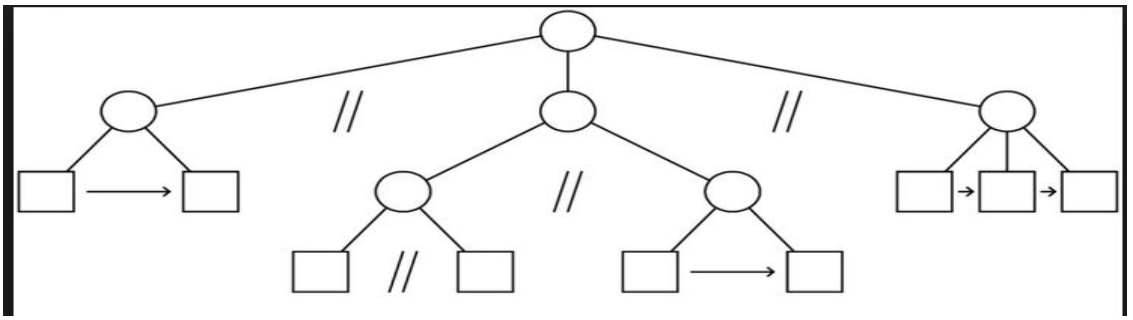
The failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is known as a **cascading schedule** or **cascading rollback** or **cascading abort**.

Premature writes:
Assume server implements abort by maintaining the "before" image of all update operations
T1 & T2 both updates account A
T1 completes its work before T2
If T1 commits & T2 aborts, the balance of A is correct
If T1 aborts & T2 commits, the "before" image that is restored corresponds to the balance of A before T2

For recoverability:
A commit is delayed until after the commitment of any other transaction whose state has been observed
(Tx's should be delayed until earlier Tx's that update the Same objects have been either committed or aborted.)

5.3 NESTED TRANSACTIONS

The nested transaction is a transaction that is created inside another transaction. A *nested transaction* is used to provide a transactional guarantee for a subset of operations performed within the scope of a larger transaction. Doing this allows you to commit and abort the subset of operations independently of the larger transaction.



Nested transactions have the following main advantages:
1. Subtransactions at one level (and their descendants) may run concurrently with other subtransactions at the same level in the hierarchy.
2. Subtransactions can commit or abort independently.

The rules to the usage of a nested transaction are as follows:

- While the nested (child) transaction is active, the parent transaction may not perform any operations other than to commit or abort, or to create more child transactions.
- Committing a nested transaction has no effect on the state of the parent transaction. The parent transaction is still uncommitted.
- Likewise, aborting the nested transaction has no effect on the state of the parent transaction.
- If the parent aborts, then the child transactions abort as well. If the parent commits, then whatever modifications have been performed by the child transactions are also committed.

5.4 LOCKS

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds ,

- Binary Locks − A lock on a data item can be in two states; it is either locked or unlocked.
- Shared/exclusive − If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

Lock compatibility

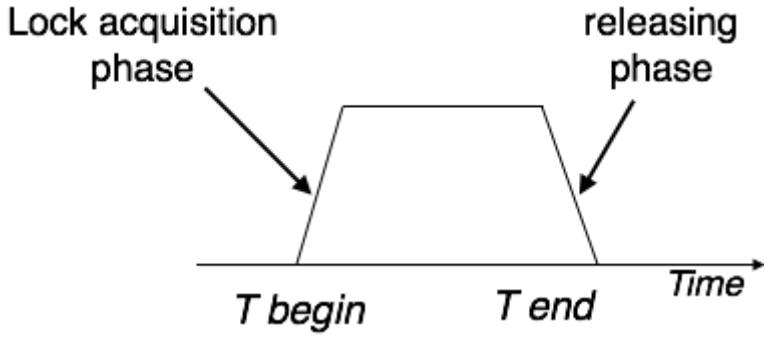
For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Types of lock protocols,

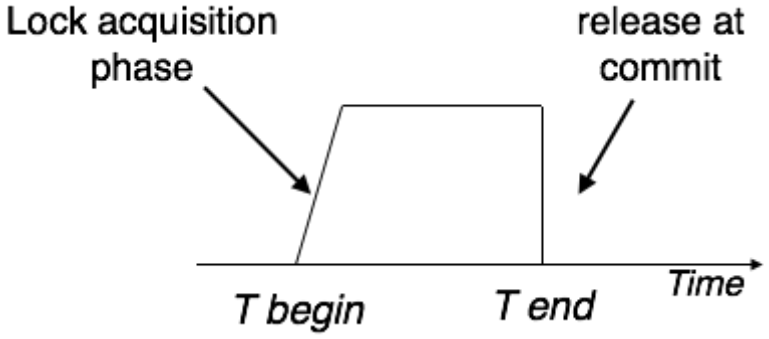
- Single phase locking
- Two phase locking- 2PL
- Strict two phase locking

Single phase : lock-based protocols allow transactions to obtain a lock on every object before a operation is performed. Transactions may unlock the data item after completing the operation.

Two phase: This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks (Growing phase). Third phase starts, the transaction cannot demand any new locks; it only releases the acquired locks (Shrinking phase).



Strict Two-Phase Locking: The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
(a)If the object is not already locked, it is locked and the operation proceeds.
(b)If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
(c)If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction

5.4.1 Lock implementation

The granting of locks will be implemented by a separate object in the server that we call the lock manager. The lock manager holds a set of locks. The methods of Lock are synchronized so that the threads attempting to acquire or release a lock will not interfere with one another. But, in addition, attempts to acquire
the lock use the wait method whenever they have to wait for another thread to release it.

All requests to set locks and to release them on behalf of transactions are sent to an instance of LockManager:

• The setLock method's arguments specify the object that the given transaction wants to lock and the type of lock.
• The unLock method's argument specifies the transaction that is releasing its locks.

5.4.2 Locking rules for nested transactions

The aim of a locking scheme for nested transactions is to serialize access to objects so that,

- Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.
- Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.

The **first rule** is enforced by arranging that every lock that is acquired by a successful subtransaction is inherited by its parent when it completes. Inherited locks are also inherited by ancestors. The top-level transaction eventually inherits all of the locks that were acquired by successful subtransactions at any depth in a nested transaction.

The **second rule** is enforced as follows:
• Parent transactions are not allowed to run concurrently with their child transactions. This means that the child transaction temporarily acquires the lock from its parent for its duration.
• Subtransactions at the same level are allowed to run concurrently.

The following rules describe lock acquisition and release:
• For a subtransaction to acquire a read lock on an object, no other active transaction
can have a write lock on that object.
• For a subtransaction to acquire a write lock on an object, no other active transaction can have a read or write lock on that object.
• When a subtransaction commits, its locks are inherited by its parent
• When a subtransaction aborts, its locks are discarded.

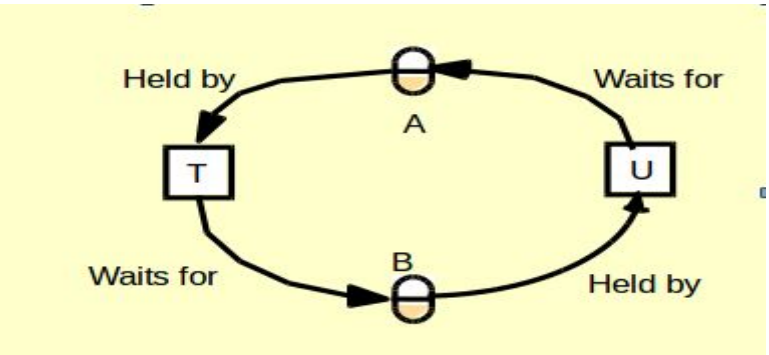
5.4.3 Deadlocks

Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.The disadvantage of Locking Deadlocks. For an example, two transaction T & U with two object A & B. T locks A and waits for U to release the locks on B. Other hand U locks B and waits for T to release the locks on A **->** Deadlock happens.
Four necessary conditions of deadlock are,

- Mutual Exclusion
- Hold and waits
- Non preemption
- circular waits

Strategies to Fight Deadlock

Deadlock Prevention: Violate one of the necessary conditions for deadlock.
Deadlock Avoidance: Have transactions declare max resources they will request, but allow them to lock at any time (Banker's algorithm)
Deadlock detection • Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.



Timeouts • Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable. However, if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes.

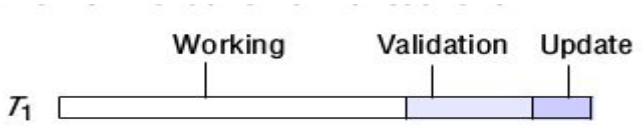
5.5 OPTIMISTIC CONCURRENCY CONTROL

Concurrency control is a concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. Optimistic concurrency control (OCC) is a concurrency control method applied to transactional systems. OCC assumes that multiple transactions can frequently complete without interfering with each other. Each transaction has the following phases:

- Working Phase
- Validation Phase
- Update Phase

Working Phase: A transaction fetches data items to memory and performs operations upon them.
Validation Phase: When the closeTransaction request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects.

Update Phase: If a transaction is validated, all of the changes recorded in its tentative versions are made permanent.



Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions – that is, any transactions that had not yet committed at the time this transaction started. To assist in performing validation, each transaction is assigned
a transaction number when it enters the validation phase. If the transaction is validated and completes successfully, it retains this number; if it fails the validation checks and is aborted. The validation rules are,

Tv	Ti	Rule
write	read	1. Ti must not read objects written by Tv
read	write	2. Tv must not read objects written by Ti
write	write	3. Ti must not write objects written by Tv and Tv must not write objects written by Ti

Two types of Validation,

- Backward Validation
- Forward Validation

Backward Validation: In backward validation, the read set of the transaction being validated is compared with the write sets of other transactions that have already committed. Therefore, the only way to resolve any conflicts is to abort the transaction that is undergoing validation. In backward validation, transactions that have no read operations (only write operations) need not be checked.

Forward Validation: In forward validation of the transaction Tv, the write set of Tv is compared with the read sets of all overlapping active tractions- those that are still in their working phase (rule 1). Rule 2 is automatically fulfilled because that active transactions do not write until after Tv has completed.

5.5.1 Comparison of forward and backward validation

conflicts-

- Forward validation allows flexibility in the resolution of conflicts. (Two methods -Defer the validation until a later time when the conflicting transactions have finished or Abort the transaction being validated).

- Backward validation allows only one choice – to abort the transaction being validated

Read Set- The read sets of transactions are much larger than the write sets.,

- Backward validation compares a possibly large read set against the old write set and it has a overhead of storing old write sets.
- Forward validation checks a small write set against the read sets of active transactions and it has to allow for new transactions starting during the validation process.

Jisy Raju