Transaction Management

Transaction: Series of actions (reads and writes) carried out by single user which accesses or changes Database, must be treated as an indivisible unit – Atomic, Logical unit of Work

Transaction transforms Database from one consistent state to another consistent state

Consistent = *Acting or done in the same way over time, especially so as to be fair or accurate*

By default, MySQL runs with *autocommit* mode enabled, i.e. all changes to a table take effect immediately, session variable

Select @@autocommit;   => 1

Transaction:

Start | Begin Transaction,

autocommit;   => 0

```
    Read (X)
    X = X-N
    Write(X)
    Read(Y)
    Y = Y + N
    Write(Y)
```

Commit | Rollback   => write out to disk

autocommit;   => 1

Four properties of a transaction (A.C.I.D. properties)

- **Atomicity** : Transaction is atomic, it is either performed entirely or not performed at all

- **Consistency** : Transaction will transform the Database from one consistent state to another

  Transaction won't violate declared system integrity constraints
  Database CAN be in a inconsistent state <u>during</u> a transaction, but at the end of the transaction the database is made consistent again.

- **Independence (Isolation)**: Transactions execute independently of one another

  Ideally, all the updates from transaction A are not visible to transaction B <u>until</u> transaction A commits

  Balance between isolation level and performance.

- **Durability (Persistence):** Effects of a successfully completed transaction permanently recorded in the database but not necessarily in <u>database</u> on disk

Data structures that implement each of the ACID properties

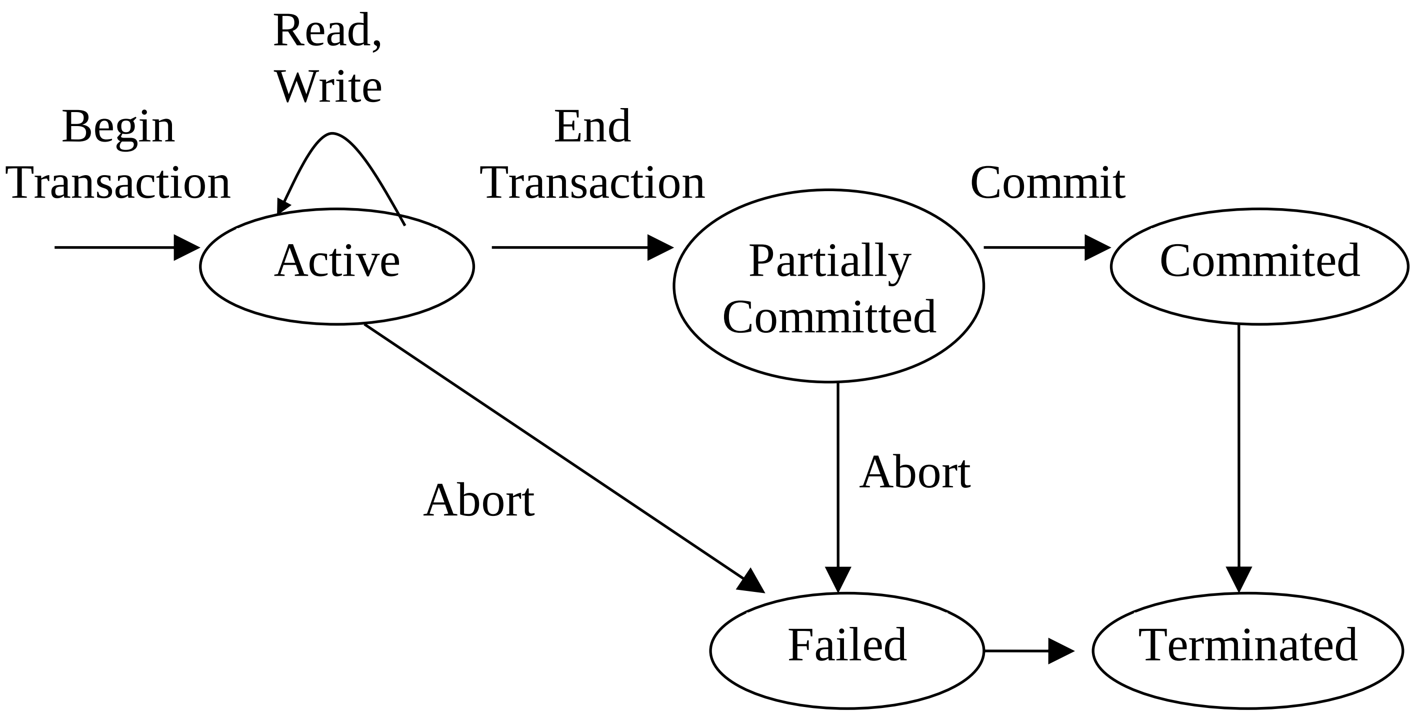
- **Atomicity** : *Autocommit, Start | Begin Tx, Commit, Rollback*
- **Consistency** : *Write buffers, Log Files, Crash recovery*
- **Independence (Isolation)**: *Autocommit, Set Isolation Level, Locking*
- **Durability (Persistence):** *Write buffers, Replication, UPS, Backups, OS, log files*

Transaction can have one of two outcomes

1. Committed: Database reaches new consistent state
2. Aborted: Database restored to previous consistent state- Rollback, Committed transaction cannot be aborted

Transaction Lify Cycle, can be in any number of states

- Begin_Transaction
- Read or Write
- End_Transaction : Check Database for consistency, see if transaction interfered with other transaction
- Commit_Transaction : Successful end of transaction
- Failed : Unsuccessful end of transaction, one of checks fails or transaction aborted



Transaction Schedules

A schedule is the order in which Transactions are executed in a system

1. Serial Schedules - no-interleaving

Transactions are executed in a serial manner i.e., no transaction starts until a running transaction has ended.

Fact: Nothing Can Go Wrong If The System Executes Transactions Serially

Highest level of isolation but poorest performance!!
Highest level of consistent data

On average, poor performance :
Excellent performance if your Tx at start of queue,
Very poor performance if your Tx at the end of the queue

Serial schedules are guaranteed to avoid interference and keep the database consistent

2. **Serializable Schedule -** interleaving

Transactions run **concurrently** and are **interleaved**

Transactions possibly executed non-serially but the overall effect on the database is equivalent to a serial schedule

Serializability: Set of transactions executing concurrently (i.e. with interleaving), overall effect on database is as though all committed transaction were executed in serial manner, i.e one after another after another

The concurrent execution of a number of transactions must be **Serially Equivalent**

Consider two transaction:

(a)		(b)	
	Read (X)		Read (X)
	X = X-N		X =X + M
	Write(X)		Write (X)
	Read(Y)		
	Y = Y + N		
	Write(Y)		

E.g. Amount (i.e. X) is 100 to start , N=20, M=30

<u>Serial</u> Schedule: Run all of T1, <u>then</u> run T2

T1	T2	T1	T2
Read (X)		100	
X = X-N		80	
Write(X)		80	
Read(Y)			
Y = Y + N			
Write(Y)			
Commit			
	Read (X)		80
	X =X + M		110
	Write (X)		**110**
	Commit		

E.g. Amount (i.e. X) is 100 to start , N=20, M=30

<u>Serializable</u> Schedule

T1	T2	T1	T2
Read (X)		100	
X = X-N		80	
Write(X)		80	
Temporarily Suspend			
	Read (X)		80
	X =X + M		110
	Write (X)		**110**
Read(Y)			
Y = Y + N			
Write(Y)			

T1	T2	T1	T2
	Read (X)		100
	X =X + M		130
	Write (X)		130
Read (X)		130	
X = X-N		110	
Write(X)		**110**	
Read(Y)			
Y = Y + N			
Write(Y)			

The two Serializable Schedules above resulted in the same consistent effect on the database (i.e. same result as a Serial Schedule) but…this might not always be the case

If several users access a database concurrently, problems may occur

Concurrency problems include:

1. **Lost Updates**: If two Tx write the <u>same</u> data, the second Tx will overwrite the first data value, thereby loosing the first Tx's write – Getting seat on flight in labs!!

2. **Dirty Read** - Tx reads a record that has not yet been committed - ***reading tentative data***.

3. **Non-Repeatable Read** - Tx re-reads <u>a particular row</u> and finds that the <u>data</u> value has changed or been deleted <u>for that one row</u>

   Not always a bad thing!!!

   When data has changed, you have to decide whether you want <u>consistent</u> data or <u>concurrent</u> data.

   If you need consistent data, then a non-repeatable read is detrimental. If you need current data, then a non-repeatable read is wonderful.

4. **Phantom Read** : In the course of a Tx, two **identical queries** are executed, and the <u>rows</u> returned by the second query are different from the first query.

   Not always a bad thing!!!

1. Lost Update Problem :

Two transactions that access the **same** Database item/s have their operations interleaved in such a way as to make the value of a Database item incorrect, an update to the database gets lost!!

E.g.

T1	T2		
Read (X)		100	
X = X-N		80	
Temporarily Suspend			
	Read (X)		100
	X =X + M		130
	Temporarily Suspend		
Write(X)		**80**	
Read(Y)			
Temporarily Suspend			
	Write (X)		**130**
Y = Y + N			
Write(Y)			

Final value of X is incorrect as T2 reads the value of X before T1 writes it to the Database

Updated value from T1(80) is lost

2. Dirty Read (Temporary Update) Problem :

A transaction updates/writes Database item and then the transaction fails for some reason but another transaction has read the updated item

E.g.

T1	T2	T1	T2
Read (X)		100	
X = X-N		80	
Write(X)		80	
Temporarily Suspend			
	Read (X)		***80***
	Dirty Read		
	X =X + M		110
	Write (X)		**110**
Read(**Y**)			
T1 Fails			

T1 fails and must change the value of X back to its old value, but T2 has read the temporarily incorrect value of X

Value of item X read by T2 is called Dirty Read as it has been created by an uncommitted transaction

Reading tentative data but….. if T1 didn't fail, this would be wonderful as T2 would be reading the most up to date data

3. Non-Repeatable Read Problem :

A transaction Reads a particular row in a table, another transaction updates this row

The original transaction re-reads the same row and gets different data

Inconsistent read

E.g.

T1	T2	T1	T2
Read (X)		**100**	
X = X-N		80	
Write(X)		80	
Temporarily Suspend			
	Read (X)		***80***
	X =X + M		110
	Write (X)		**110**
Read(X)		**110**	

Reads a particular row and finds that the data has been changed or deleted for that one row, data was 100, now 110

Not always a bad thing!!!

4. Phantom Read:

T1 executs the same query twice and a different set of rows were returned the second time

E.g.

T1	T2

Select * from users
where age between 10 and 30;
 	
return **2** records.
Temporarily Suspend	

	Insert into users
	values (3, 'Bob', 27);
	
	Commit;

Select * from users
where age between 10 and 30;
 	
return **3** records.	

Not always a bad thing!!!

Each of the above problems may or may not be an issue for your database

One of the considerations is whether consistent data is your priority or whether concurrent data is your priority

Independence (Isolation)

Ideally, one Tx cannot interfere with another Tx.

Isolation level

- Defines when changes made by one Tx become visible to other concurrent Tx
- Balance Between Performance and Reliability, Consistency
- Various levels are not "bad" or "good", it depends on what your application requires in terms of isolation.

Various Transaction Isolation levels (**lowest** to **highest**)

1. **Read Uncommitted**
2. **Read Committed**
3. **Repeatable Read** - **Default for Innodb**
4. **Serrializable - Default for MySQL**

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Prevented	Allowed	Allowed
Repeatable Read	Prevented	Prevented	Allowed
Serializable	Prevented	Prevented	Prevented

1. **Read Uncommitted** –
   - Tx B will see the changes that Tx A makes prior to TxA commiting
   - Great for concurrency / performance provided …. Tx A doesn't do a rollback
   - Possibility of **Dirty Reads, Phantom reads** and **Non-Repeatable Reads**

2. **Read Committed** –
   - Tx B will only see the changes that Tx A makes after TxA commits
   - Possibility of **Non-Repeatable Reads** and **Phantom Reads -** Data getting changed in current transaction by other transactions.
   - Does not allow dirty reads

3. **Repeatable Read - Default for Innodb**
   - **Snapshot** – Version | Copy of database
   - Guaranteeds to see same data for Selects
   - Possibility of **Phantom Reads -** Data gets changed in current transaction by another transactions doing Updates/ Insert/Delete – Phantom Read
   - https://bugs.mysql.com/bug.php?id=63870

4. **Serializable - Default for SQL**
   - **Select** gets converted to Read Lock,
   - **Insert /Update /Delete** gets converted to Wr lock, Database will lock data until Tx commits
   - Lots of locking – Big performance hit!!
   - Performs all transactions in sequence, serially

```
Begin;
    Select * From T;
    Waitfor Delay '00:01:00'
    Select * From T;
Commit;
```

- **READ UNCOMMITTED**, second SELECT may return **<u>different</u>** data. A concurrent transaction may update the record, delete it, insert new records. The second select will **see** the very latest data from all <u>active</u> transactions.

- **READ COMMITTED**, second SELECT may return **<u>different</u>** data. A concurrent transaction may update the record, delete it, insert new records. The second select will always **see** the very latest data from all <u>committed</u> transactions.

- **REPEATABLE READ**, second SELECT is guaranteed to see the same rows as the first select

  New rows may be added by a concurrent transaction during the one minute, but the existing Tx cannot see them.

- **SERIALIZABLE**, second SELECT is guaranteed to see exactly the same rows as the first Select

  Select automatically acquires read lock in this mode, so no row can be modified/deleted by a concurrent transaction.

Repeatable Read

Consistent read view or consistent snapshot.

Time	T1	T2
1	**Start Tx;**	
2	SnapShot	**Start Tx;**
3	Select * from T;	Snapshot
4	{empty set}	
5		Insert into T values (1, 2);
6	Select * from T;	
7	{empty set}	
8		Select * from T;
9		{1,2}
10		**Commit;**
11	Select * from T;	
12	{empty set}	
13		
14	**Commit;**	
15	Select * from T;	
16	{1 ,2}	

Repeatable Read

Initial value of field x in table is 10

Time	Tx A	Tx B
1	Start Tx;	
2	SnapShot	Start Tx;
	Select x from table; (10)	SnapShot
4	update Table set x=x+10; (20)	
5		select x from Table; (10)
6	Select x from table; (20)	
7	Commit;	
8		select x from Table; (10)
9		Commit;
10		select x from Table; (20)

Repeatable Read
Example of Phantom Read

Time	Tx A	Tx B
1	Start Tx;	
2	SnapShot	Start Tx;
3	Select * from table; (1,1),(2,2),(3,3)	SnapShot
4		select * from Table; (1,1),(2,2),(3,3)
5		Insert into table values (4,4);
6	Select * from table; (1,1),(2,2),(3,3)	
7		Commit;
8	Select * from table; (1,1),(2,2),(3,3)	

9	**Update** table Set field2=5;
	Four rows changed;	

10	Select * from table; (1,**5**),(2,**5**),(3,**5**), **(4,5)** **Phantom Read**	

Read Committed

Set Session Transaction Isolation Level Read Committed;

No Snapshot here

Example of Non-Repeatable Read

Time	Tx A	Tx B
1	Start Tx;	
2	select * from Table; (1,1),(2,2),(3,3)	Start Tx;
3	**Select * from table where field1=1;** **(1,1)**	
4		select * from Table; (1,1),(2,2),(3,3)
5		**Update** table Set field2=5 where field1=1;
6		Commit;
7	**Select * from table where field1=1;** **(1,5)** **Non-Repeatable Read**	

Concurrency Control

Concurrency Control is managing simultaneous transactions in a database without each transaction interfering with one another

Coordinating the actions of transactions that

- Operate in parallel
- Access **shared** data
- **Potentially** interfere with each other

in order to lead to better transaction throughput and response time

DBMS <u>behaves</u> as if it executes operations sequentially, i.e., one at a time, one after another but …..DBMS typically will execute operations *concurrently* i.e. executing more than one operation at once.

However the DBMS executes transactions, the final and overall effect on the database must be the same as if sequential execution occured

We will look at three concurrency control techniques :

1. Locking
2. Versioning
3. Multi-version concurrency control (MVCC)

1. Locking

Traditional concurrency control

Possible to lock data at different levels:

- Database : Entire Database is locked and unavailable
  Used for system backups

- Table : Entire Table is locked
  Used for general updates

- Block or Page : Physical Page is locked
  Generally not desirable

- Row Level : Only requested row is locked
  Most common implementation
  All other rows free,
  But overhead at run-time

- Field Level : Only requested field (column) is locked
  Used when an update is to one or two fields in a table
  Rarely used at they require considerable overhead

Locking level also called Granularity

- Fine granuality – Field Row, Page – lots of locks, a performance overhead, more concurrency

- Coarse granuality – Database Table – fewer locks but less overhead, less concurrency

InnoDB supports Multiple GranualityLocking which permits coexistence of row locks and table locks

Table Intention Locks are used by a transaction to indicate which type of lock (shared or exclusive) it will require later for a row in that table.

There are two types of intention locks used in InnoDB:

- Intention shared (IS): Transaction intends to set a read lock.
- Intention exclusive (IX): transaction intends to set an write lock.

Intention locks rarely block (only LOCK TABLE. ….WRITE)

Multiple transactions can acquire IX locks. These locks are at the table-level, not the row-level. An IX lock means that the thread holding it intends to update some rows *somewhere* in the table.

IX locks are only intended to block full-table operations

The idea of intention locks is to allows the lock requests to be resolved in order, but not blocking locks and operations that are compatible

Whether a lock is granted or not can be summarised as follows:

- An X lock is not granted if any other lock (X, S, IX, IS) is held.

- An S lock **is not** granted if an X or IX lock is held.
  An S lock **is** granted if an S or IS lock is held.

- An IX lock **is not** granted if in X or S lock is held.
  An IX lock **is** granted if an IX or IS lock is held.

- An IS lock **is not** granted if an X lock is held.
  An IS lock **is** granted if an S, IX or IS lock is held.

	X	***IX***	***S***	***IS***
X	Conflict	Conflict	Conflict	Conflict
S	Conflict	Conflict	Compatible	Compatible
IX	Conflict	Compatible	Conflict	Compatible
IS	Conflict	Compatible	Compatible	Compatible

Two types of locks :

- Shared locks (S Locks, Read Locks)

Allow other transaction to read but not update record

Placing a shared lock on a record prevents another user from placing an exclusive lock on that record

Possible for more than one transaction to hold read locks on same data item

- Exclusive Lock (Write Lock, X Lock)

Prevents another transaction from reading and writing a record until it is unlocked

Placing an exclusive lock on a record prevents another Tx from placing any lock on that record

Row Locks are automatically obtained by DBMS, without user involvement

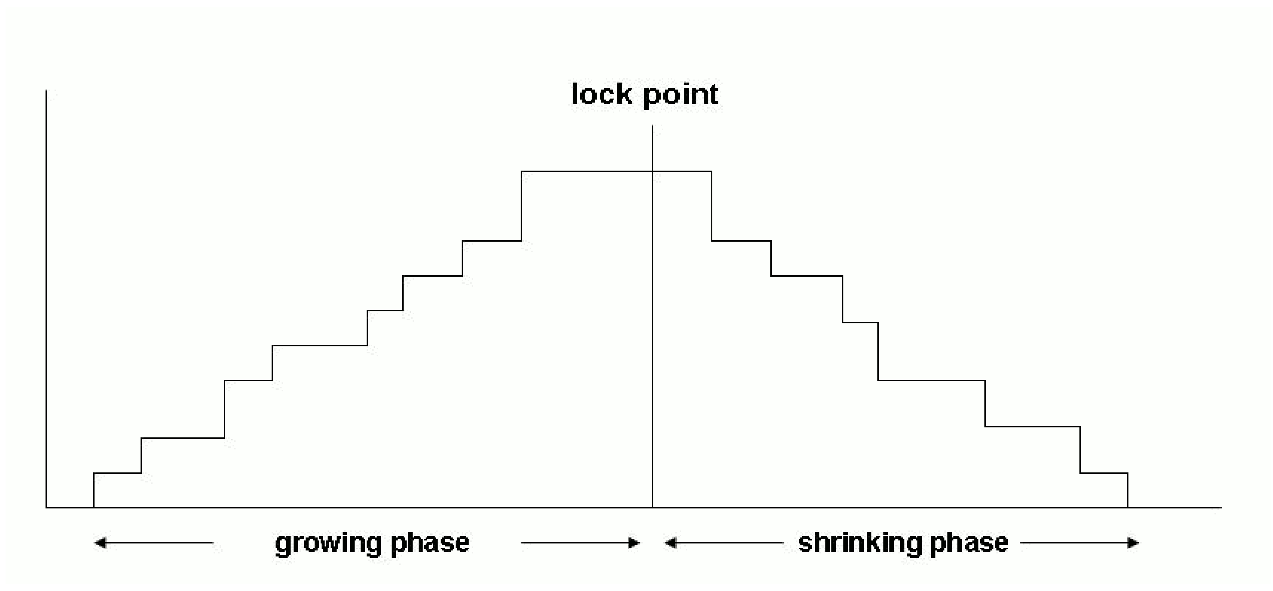
Possible, under certain circumstances, to allow 'lock conversion/upgrading ' i.e. upgrade read lock to write lock or downgrade write lock to read lock

Two-Phase Locking (2PL)

Two phases:

1. Growing Phase where transaction acquire locks
2. Shrinking Phase where transaction release locks

- If T wants to read an object, first obtains an S lock.
- If T wants to modify an object, first obtains X lock.
- T can release lock during T
- But if T releases any lock, it cannot acquire new locks!



Possibility of **lost updates** as once a lock released, data item it can be changed by another transaction before the first transaction commits or Rollbacks.

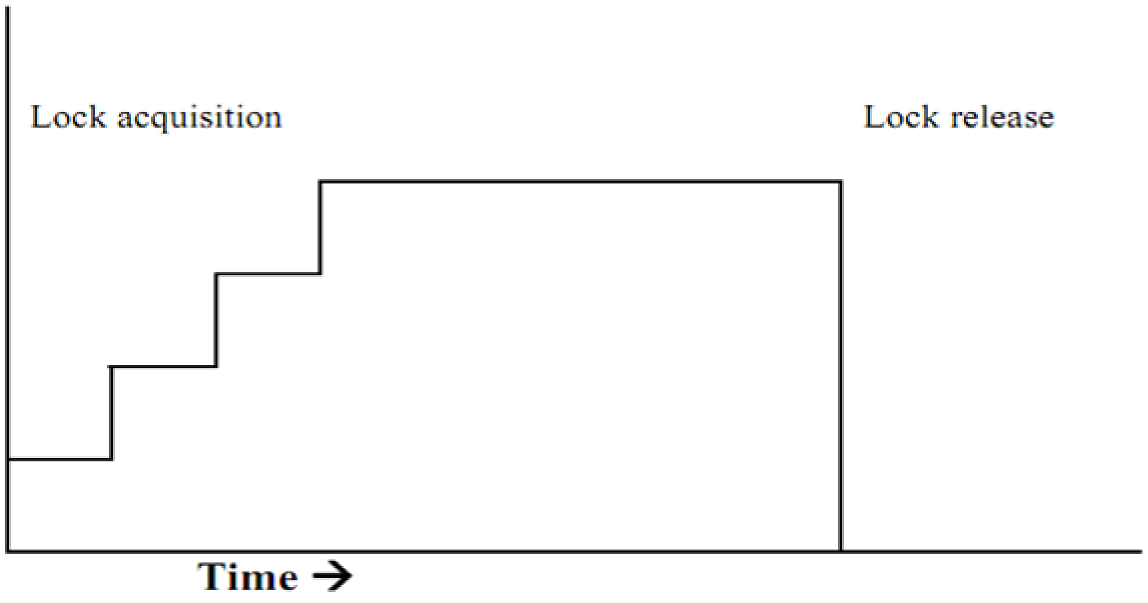
If lock upgrading allowed, upgrading must be performed during expansion phase and downgrading performed during shrinking phase

Lead to development of Strict 2PL

Strick Two-Phase Locking (2PL)

Well Formed Transaction

- If T wants to read an object, first obtains an S lock.
- If T wants to modify an object, first obtains X lock.
- Lock acquisition performed at start of T – should prevent deadlock
- <u>Hold all locks until end of transaction.</u>
- Release all locks together at end of transaction



Drawbacks of strict 2PL is **restricts concurrency** as it locks the item until <u>after</u> until TX commits

No problem then with **Lost Updates**

The locking phase usually implemented as a loop where all the locks are acquired one by one.

If any lock is not acquired on the first attempt, the algorithm releases all the locks it had acquired, and starts to try to get all the locks again - "back-off and re-try"

Can lead to **Transaction Starvation**, if a single Transaction never acquires all the locks needed for it to continue execution.

- It occurs if the waiting scheme for locked items is unfair, giving priority to some transactions over others.
- Starvation happens if same transaction is always choosen as victim.
- Tx never progress.

Starvation Avoidance:

- Switch priorities so that every thread has a chance to have high priority.
- Use FIFO order among competing request.

Deadlock

Two or more Tx are waiting for each other to finish

Tx’s are unable to proceed because each is waiting for the other to do something.

Assume two customers accessing a joint account

T1	T2
Request **S** Lock on Balance	
Read Balance	
	Request **S** Lock on Balance
	Read Balance
Request **X** lock on Balance (denied)	
	Request **X** lock on Balance (denied)
(wait)	
	(wait)

Above sequence results in a situation called **Deadlock**

T1 is waiting for T2 to release the S lock while T2 is waiting for T1 to release the S lock

Impasse that results when two or more transactions have locked a resource and each must wait for the other to unlock that resource

Overcome this problem using:

- **Deadlock Prevention** - User must lock ALL records at the beginning of a transaction

  But, difficult to predict what records will be required

  Ties up resources for a long time

- **Deadlock Resolution** - Allow deadlock to occur but build in a mechanism for detecting and breaking deadlock.

  Generally results in rollback of some transaction

Innodb has automatic deadlock detection algorithm

Will rollback last transaction to receive lock involved in deadlock

If lock placed by a different storage engine, e.g. MyISAM,   MySQL will detect long transactions and rollback long transactions

Starvation	Deadlock
Starvation happens if same transaction is always choosen as victim.	Two or more Transactions are waiting for each other to finish
It occurs if the waiting scheme for locked items is unfair, giving priority to some transactions over others.	Transactions are unable to proceed because each is waiting for the other to do something.
Avoidance: •Switch priorities so that every thread has a chance to have high priority. •Use FIFO order among competing request.	Avoidance: •Acquire locks in a predefined order. •Acquire locks immediately before starting.
Tx never progress.	Tx's are waiting for each other.

2. Versioning

Locking referred to as **pessimistic** concurrency control as each time record required, DBMS locks record so that other programs cannot use it

Versioning takes **Optimistic** approach in assuming that users will not request same data, at same instant in time. If they do, users may only need to read data, not write the data

In strict Versioning, No Locking of data required

Database or table supports different views for the data depending on when access begins.

Other common terms for this are “time travel,” “copy on write,” or “copy on demand.”

Each transaction is restricted to a view of the Database as of the time that transaction started

At commit time, check performed to determine whether conflict occurred, oldest transaction is given priority

If conflict, all newer transactions notified of the conflict, rolled back and restarted

But, rollbacks *should* be rare
Increased performance improvement due to greater concurrency

Assume two customers accessing a joint account

T1	T2
Read Balance (1,000)	
	Read Balance (1,000)
Withdraw 200	
	Withdraw 500
Commit	
	Commit Return to database Conflich Identified
	… Rollback
	Restart Transaction

At commit time, Database discovers that T2's update conflicts, transaction is aborted and restarted using new updated balance

3. Multi-Version Concurrency Control (MVCC)

Until recently, nearly all databases used **pessimistic** concurrency model that locked all or portions of the database while a single transaction updated it

MVCC is an **optimistic** concurrency model

Goal being to combine the best properties of a versioning databases with traditional two-phase locking.
Used my Oracle, PostGre, MySQL, SQLServer, IBM DB2, etc

Combination of versioning and Locking

Locking is implemented mainly at the row level

MVCC in MySQL

Innodb keeps information about old versions (values) of rows in a *undo log*

InnoDB adds three fields to each row of each tabe stored in the database.

1. TR_ID : Id of Tx that modified the row.
2. PTR : Points to previous version of that row inside the undo log
3. ROW_ID : Counter that increases as new rows are inserted.

InnoDB storage engine uses primarily row-level locking for improved concurrency – at the price of greater complexity

Good concurrency as a given table can be read and modified by different clients at the same time.

Innodb does not lock the tables in unless explicitly requested via LOCK TABLES command.

InnoDB normally acquires only intentional shared (IS) or intentional exclusive (IX) modes.

Row-level concurrency properties are as follows:

- Different clients can <u>read</u> the same rows simultaneously.
- Different clients can <u>modify</u> **different** rows simultaneously.
- Different clients cannot modify the same row at the same time. If one Tx modifies a row, otherTx cannot modify the same row until the first Tx completes, i.e. releases the excusive lock

When another transactions sees this update all depends on the Transaction Isolation evel

MySQL locks

MyISAM

- Table level locking e.g.

```
LOCK TABLES  emp READ  | WRITE
      Update emp set salary = salary*1.2
     where deptno =30;
UNLOCK TABLES;
```

Innodb :

InnoDB explicitly handles lock, user has nothing to do

- Row level locking , e.g.

```
SELECT ..... from emp FOR UPDATE
  ........
Commit | Rollback
```

Exclusive lock on the rows selected

```
SELECT ..... from emp LOCK IN SHARE MODE
  .....
Commit | Rollback
```

Lots of Read locks on the rows selected

Remember plain Select sets no locks