Transactions and Recovery

Jianjun Chen

Contents

- Transactions
- Recovery
  - System and Media Failures
- Concurrency
  - Concurrency problems
- For more information:
  - Connolly and Begg chapter 22

Transactions: Definition

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

Transaction Support In MySQL

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Transaction Support In MySQL

```
1  START TRANSACTION;
2  SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
3  UPDATE table2 SET summary=@A WHERE type=1;
4  COMMIT;
```

- More information can be found at :

https://dev.mysql.com/doc/refman/8.0/en/commit.html

Transactions

- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest
- Transactions are the unit of recovery, consistency, and integrity as well
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Example of transaction

Transfer £50 from account A to account B:

1. Read(A)
2. A = A - 50
3. Write(A)
4. Read(B)
5. B = B+50
6. Write(B)

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

Atomicity and Consistency

- Atomicity:
  - Transactions are atomic: they don't have parts (conceptually)
  - So a transaction can't be executed partially.
  - "One transaction should be treated as one single action"
- Consistency
  - Transactions take the database from one consistent state into another.
  - In the middle of a transaction the database might not be consistent

Isolation and Durability

- Isolation
  - The effects of a transaction are not visible to other transactions until it has completed
  - From outside the transaction has either happened or not
  - To me this actually sounds like a consequence of atomicity…
- Durability
  - Once a transaction has completed, its changes are made permanent
  - Even if the system crashes, the effects of a transaction must remain in place

The Transaction Manager

The transaction manager enforces the ACID properties

- It schedules the operations of transactions
- COMMIT and ROLLBACK are used to ensure atomicity
- Locks are used to ensure consistency and isolation for concurrent transactions
- A log is kept to ensure durability in the event of system failure

COMMIT and ROLLBACK

- COMMIT signals the successful end of a transaction
  - Any changes made by the transaction should be saved
  - These changes are now visible to other transactions
  - Writes the data change in the memory to the DB file.

- ROLLBACK signals the unsuccessful end of a transaction
  - Any changes made by the transaction should be undone
  - It is now as if the transaction never existed

Recovery

- Transactions should be durable, but we cannot prevent all sorts of failures:
  - System crashes
  - Power failures
  - Disk crashes
  - User mistakes
  - Sabotage
  - Natural disasters

- Prevention is better than cure
  - Reliable OS
  - Security
  - UPS and surge protectors
  - RAID arrays
- However, we cannot protect against everything

Media Failures

- System failures are not too severe
  - Only information since the last checkpoint is affected
  - This can be recovered from the transaction log

- Media failures (disk crashes etc) are more serious
  - The data stored to disk is damaged
  - The transaction log itself may be damaged

Backups

- Backups are needed to recover from media failure
  - The transaction log and entire contents of the database is written to secondary storage (often tape)
  - Time consuming, and often requires down time
- Backups frequency
  - Frequent enough that little information is lost
  - Not so frequent as to cause problems
  - Every day (night) is common
- Backup storage

Recovery from Media Failure

1. Restore the database from the last backup
2. Use the transaction log to redo any changes made since the last backup

If the transaction log is damaged, you can’t do step 2

- Store the log on a separate physical device to the database
- The risk of losing both is then reduced

Concurrency

- Large databases are used by many people
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Long transactions (e.g. backups) will make others wait for long periods

Concurrency Problems

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several sorts of problems
  - Lost updates
  - Uncommitted updates
  - Incorrect analysis
- All arise because isolation is broken

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

- T1 and T2 read X, both modify it, then both write it out
  - The net effect of T1 and T2 should be no change on X
  - Only T2's change is seen, however, so the final value of X has increased by 5

Uncommitted Update

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

- T2 sees the change to X made by T1, but T1 is rolled back
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

Inconsistent analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

- T1 doesn't change the sum of X and Y, but T2 sees a change
  - T1 consists of two parts – take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

Concurrency

Concurrency control, Serialisability, Locks, Deadlocks

Schedules

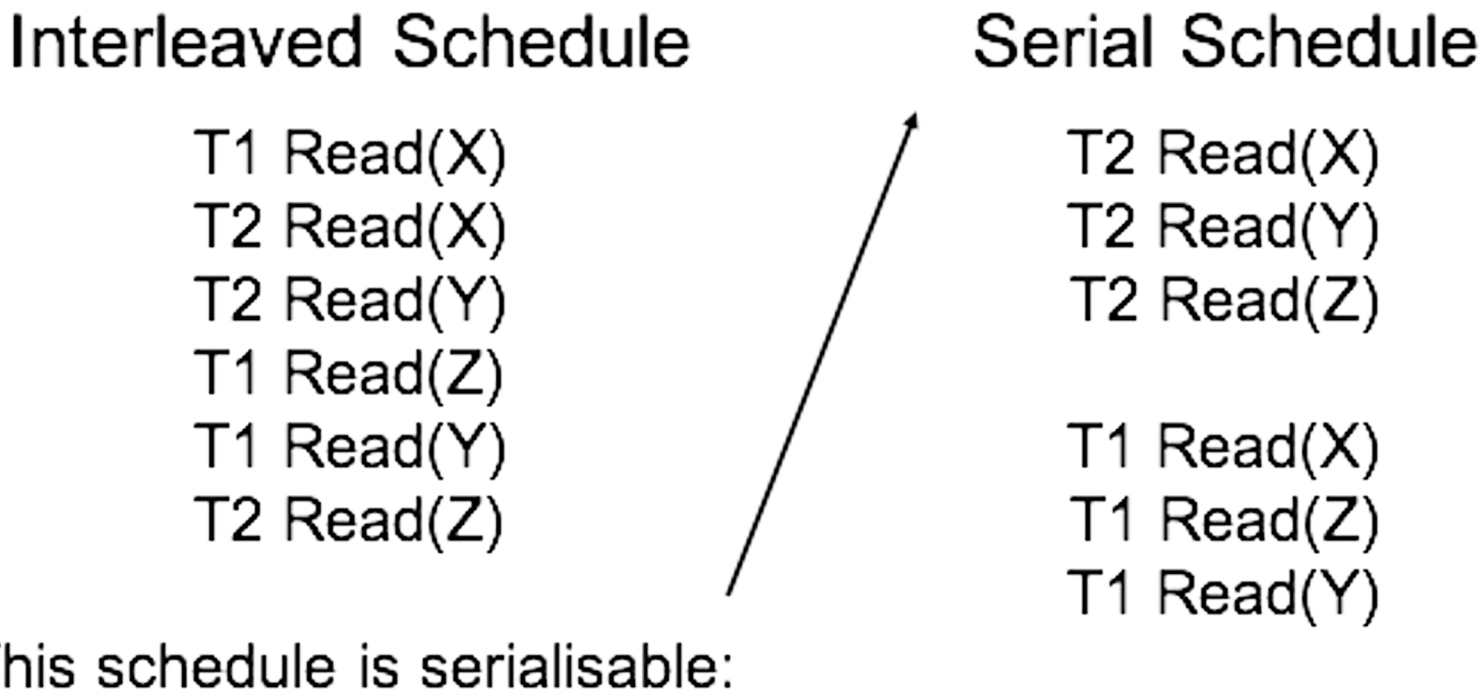
- A **schedule** is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
  - A **serial schedule** is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)
  - Non-serial schedule: operations from transactions are interleaved

Serialisability

- A non-serial schedule is **serialisable** if it produces the same results as some serial execution.

- The objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another

Serial and Serialisable



This schedule is serialisable:

Order of Read and Write

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.
- If one transaction writes a data item and another either reads or writes the same data item, **the order of execution** is important (for preventing interference). They have **conflict**

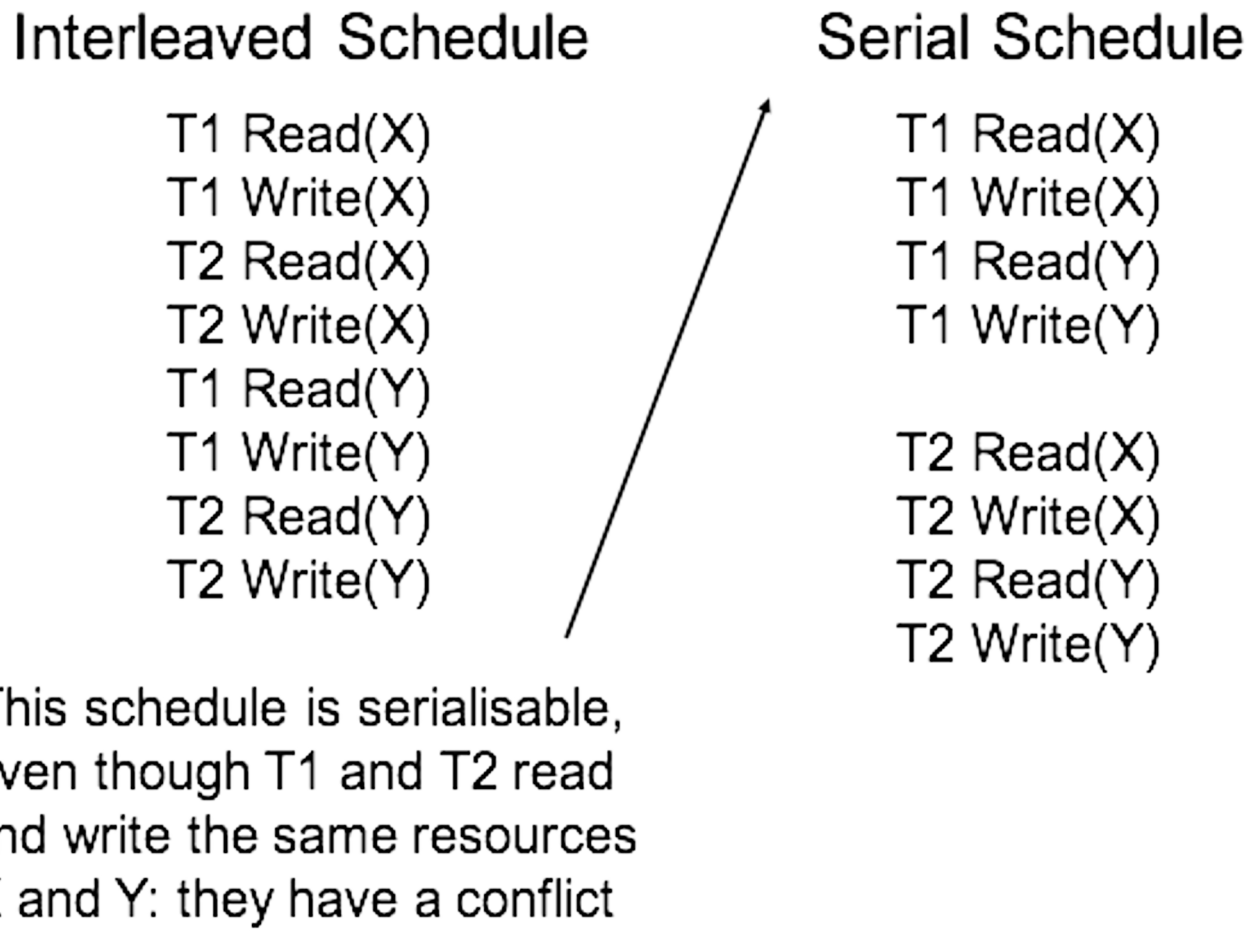
Conflict Serialisability

A schedule is **conflict serialisable** if transactions in the schedule have a conflict but the schedule is still serializable (equivalent to some serial schedule).

Previous definitions:

- Schedule: sequence of operations from multiple transactions.
- Serial schedule: One transaction after another.
- Serialisable: interleaved but still has same effect as serial schedule.

Conflict Serialisable Schedule



Conflict Serialisability

- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule

- Important questions:
  - Given a schedule, how to determine whether a schedule is conflict serialisable?
  - How to construct conflict serialisable schedules?

Precedence Graphs

To determine if a schedule is conflict serializable, we use a precedence graph.

Steps:

1. Create a node for each transaction.
2. Create a directed edge $T_i \rightarrow T_j$
   - If T_j reads the value of an item written by T_i.
   - If T_j writes a value into an item after it has been read by T_i.
   - If T_j writes a value into an item after it has been written by T_i.

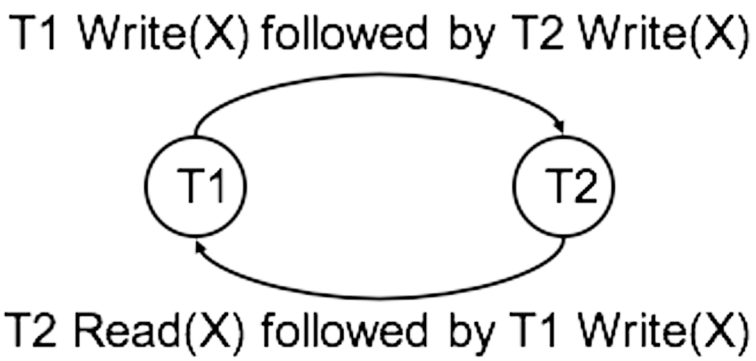
Precedence Graphs

- If an edge T*i* → T*j* exists in the precedence graph for S, then in any serial schedule S′ equivalent to S, T*i* must appear before T*j*.
- If the precedence graph contains a cycle the schedule is not conflict serializable.
- The precedence graph can help us identify whether lock is needed for a certain resource. Lock prevents other transaction from accessing the same resource.

Precedence Graph Example

- The lost update schedule has the precedence graph:

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT



Precedence Graph Example

- No cycles: conflict serialisable schedule

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

T1 reads X before T2 writes X and
T1 writes X before T2 reads X and
T1 writes X before T2 writes X

Precedence Graph Explanation

- If we have T1 -> T2, the schedule requires that T2 should be done after T1.
- If we have T2 -> T1, the schedule requires that T1 should be done after T2.

- If both edges exists, that's a conflict on interleaving that we cannot solve.
  - If T1 is requested before T2, then the precedence of T2

Locking

Locking

- Locking is a procedure used to control concurrent access to data (to ensure serialisability of concurrent transactions)
- In order to use a ‘resource’ (table, row, etc) a transaction must first acquire a lock on that resource
- This may deny access to other transactions to prevent incorrect results

Two types of locks

- Two types of lock
  - Shared lock (S-lock or read-lock)
  - Exclusive lock (X-lock or write-lock)
- Read lock allows several transactions simultaneously to read a resource
  - but no transactions can change it at the same time
- Write lock allows one transaction exclusive access to write to a resource.
  - No other transaction can read this resource at the same time.
- The lock manager in the DBMS assigns locks and records them in the data dictionary

Locking

- Before reading from a resource a transaction must acquire a **read-lock**
- Before writing to a resource a transaction must acquire a **write-lock**
- Locks are released on commit/rollback

Locking

- A transaction may not acquire a lock on any resource that is write-locked by another transaction
- A transaction may not acquire a write-lock on a resource that is locked by another transaction
- If the requested lock is not available, transaction waits

Two-Phase Locking

- A transaction follows the ***two-phase locking protocol*** **(2PL)** if all locking operations precede the first unlock operation in the transaction:

- **Growing phase** where locks are acquired on resources
- **Shrinking phase** where locks are released

Example

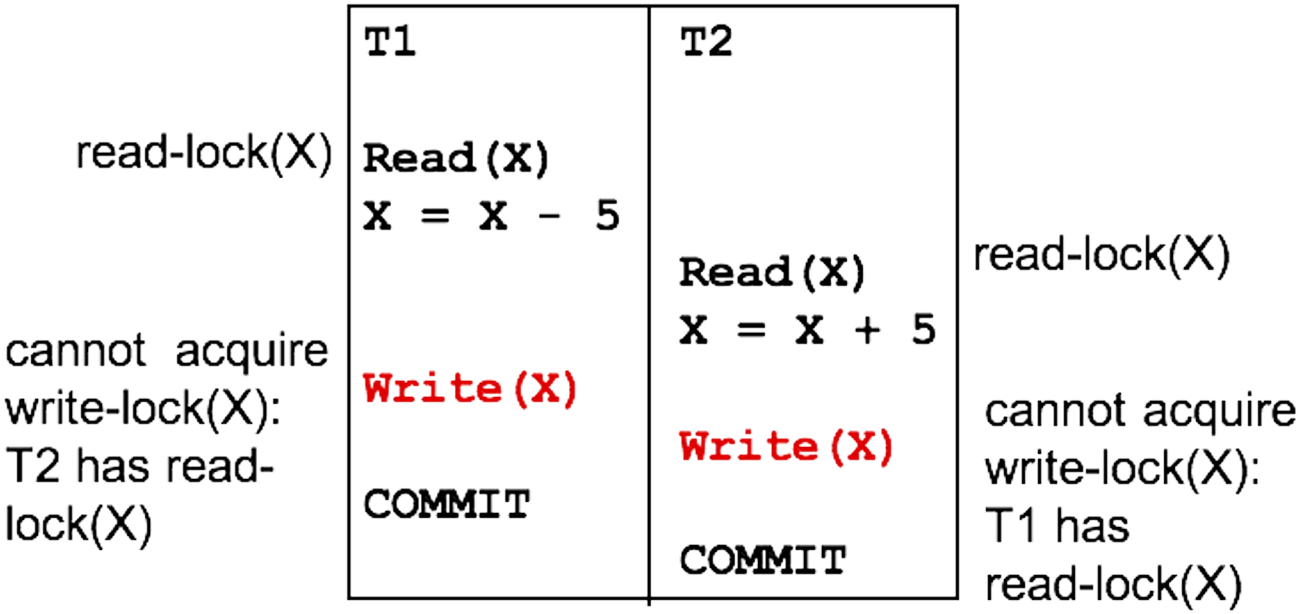
- T1 follows 2PL protocol
  - All of its locks are acquired before it releases any of them
- T2 does not
  - It releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

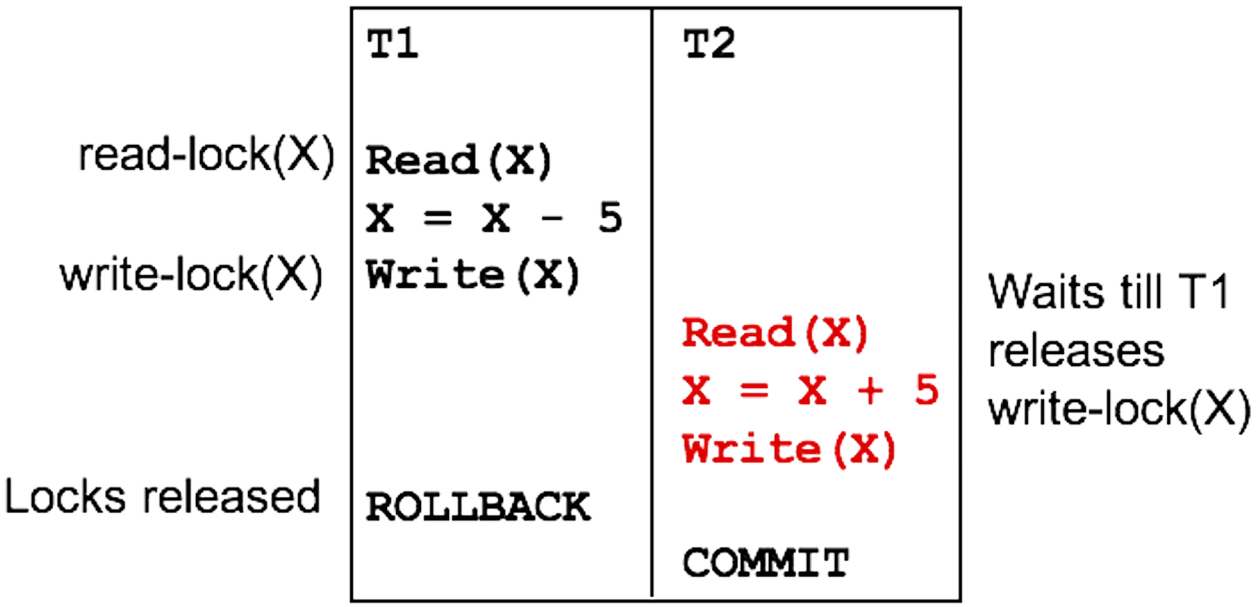
Serialisability Theorem

Any schedule of two-phased transactions is conflict serialisable

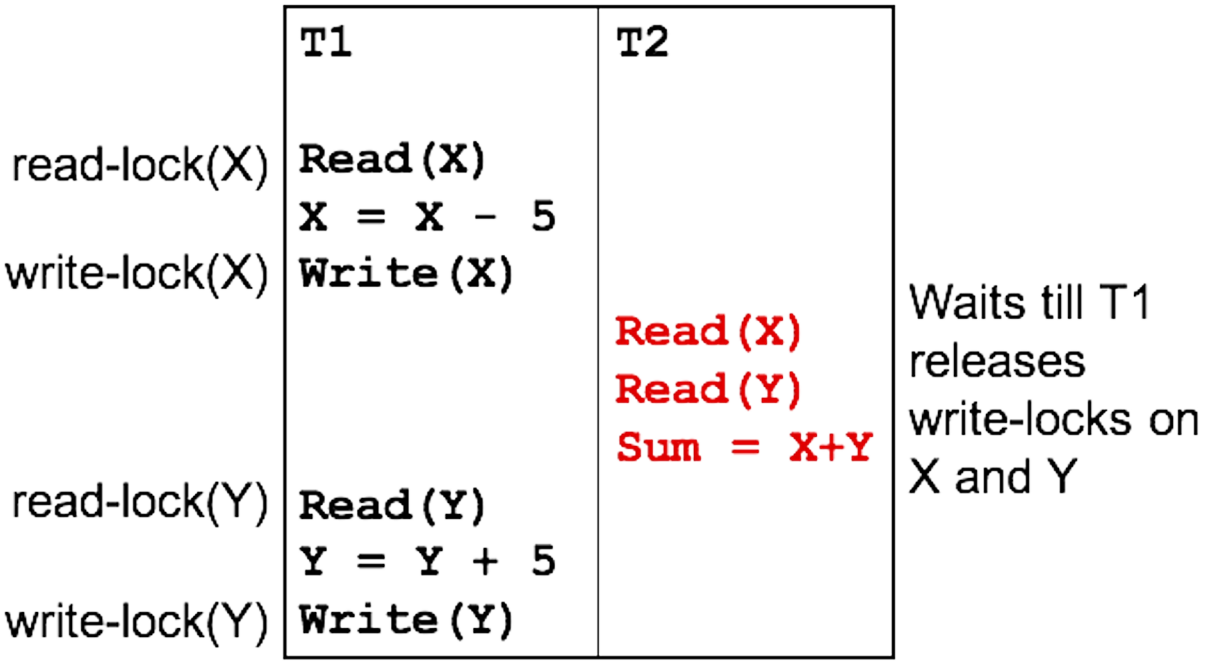
Lost Update can't happen with 2PL



Uncommitted Update cannot happen with 2PL



Inconsistent analysis cannot happen with 2PL



Question: What will happen if Read(Y) from T2 is before Read(x)?

Locks in MySQL

- https://www.oreilly.com/library/view/mysql-reference-manual/0596002653/ch06s07.html

Deadlocks

And Prevention Methods.

Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example:
  - T1 has a lock on X and is waiting for a lock on Y, and
  - T2 has a lock on Y and is waiting for a lock on X.

- Given a schedule, we can detect deadlocks which will happen in this schedule using a **wait-for graph (WFG)**.

Precedence/Wait-For Graphs

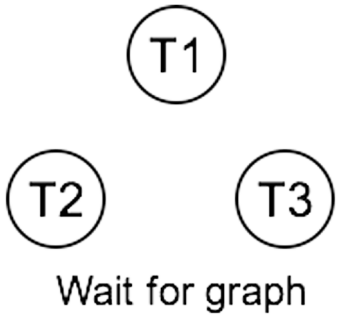
Creating Wait-for Graph

- Each transaction is a vertex
- Arcs from T1 to T2 if T1 is waiting to lock an item that is currently locked by T2.

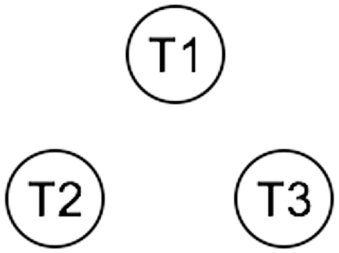
Deadlock exists if and only if the WFG contains a circle.

Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



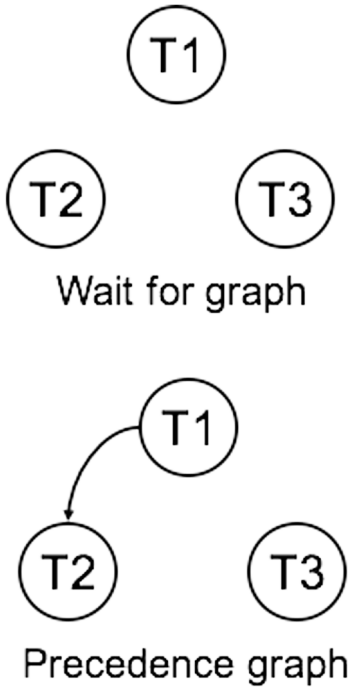
Wait for graph



Precedence graph

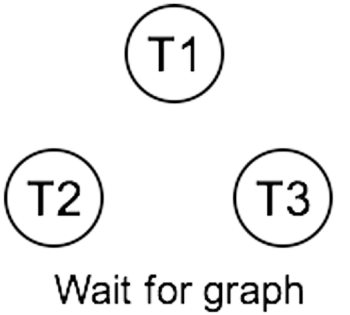
Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)

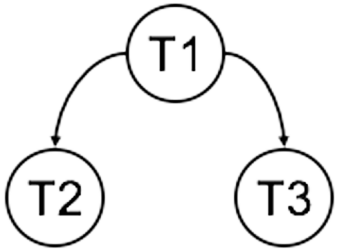


Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



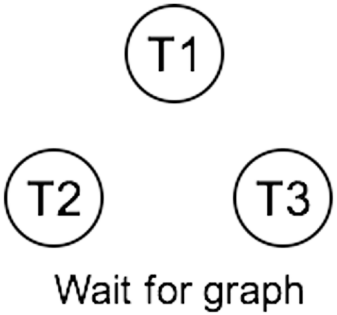
Wait for graph



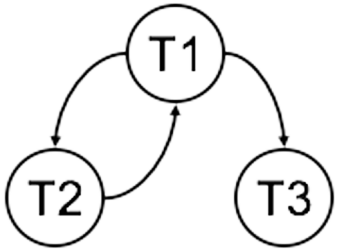
Precedence graph

Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



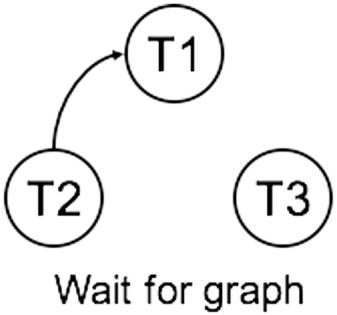
Wait for graph



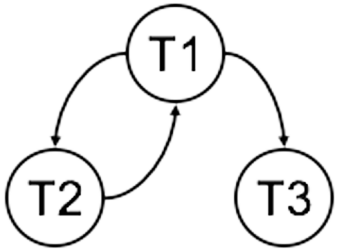
Precedence graph

Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



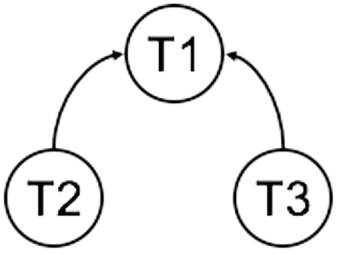
Wait for graph



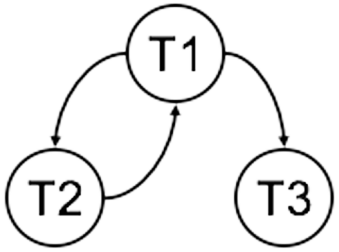
Precedence graph

Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z) read-lock(Z)
- T3 Write(Z) write-lock(Z)
- T1 Read(Y) read-lock(Y)
- T3 Read(X) tries read-lock(X)
- T1 Write(Y)



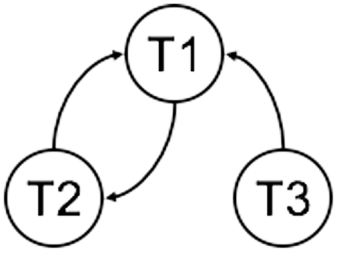
Wait for graph



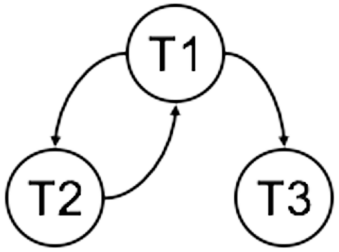
Precedence graph

Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z) read-lock(Z)
- T3 Write(Z) write-lock(Z)
- T1 Read(Y) read-lock(Y)
- T3 Read(X) tries read-lock(X)
- T1 Write(Y) tries write-lock(Y)



Wait for graph



Precedence graph

Deadlock Prevention

- Deadlocks can arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether
- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - Low ‘lock utilisation’ - transactions can hold on to locks for a long time, but not use them much

Deadlock Prevention: Resource Reordering

- We impose an ordering on the resources
  - Transactions must acquire locks in this order
  - Transactions can be ordered on the last resource they locked

- This prevents deadlock
  - If T1 is waiting for a resource from T2 then that resource must come after all of T1's current locks
  - All the arcs in the wait-for graph point 'forwards' - no cycles

Example of resource ordering

- Suppose resource order is:
  - X < Y
- This means, if you need locks on X and Y, you first acquire a lock on X and only after that a lock on Y
  - even if you want to write to Y before doing anything to X

- It is impossible to end up in a situation when T1 is waiting for a lock on X held by T2, and T2 is waiting for a lock on Y held by T1.