Transaction Management

Concurrency control

Connolly & Begg. Chapter 19. Third edition

Concurrency Control

- Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for Concurrency Control

Three examples of potential problems caused by concurrency:

- Lost update problem
- Uncommitted dependency problem
- Inconsistent analysis problem.

Lost Update Problem

Successfully completed update is overridden by another user.

Example:

- T_1 withdraws £10 from an account with bal_x, initially £100.
- T_2 deposits £100 into same account.
- Serially, final balance would be £190.

Lost Update Problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Loss of T_2's update!!

This can be avoided by preventing T_1 from reading bal_x until after update.

Uncommitted Dependency Problem

Occurs when one transaction can see intermediate results of another transaction before it has committed.

Example:

- T_4 updates bal_x to £200 but it aborts, so bal_x should be back at original value of £100.
- T_3 has read new value of bal_x (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

Uncommitted Dependency Problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Problem avoided by preventing T3 from reading bal_x until after T4 commits or aborts.

Inconsistent Analysis Problem

Occurs when transaction reads several values but second transaction updates some of them during execution of first.

Example:

- T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, T5 has transferred £10 from bal_x to bal_z, so T6 now has wrong result (£10 too high).

Inconsistent Analysis Problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

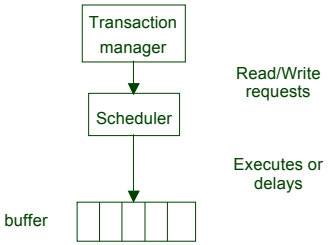
Problem avoided by preventing T6 from reading bal_x and bal_z until after T5 completed updates.

Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
- Possible solution: Run all transactions serially.
- This is often too restrictive as it limits degree of concurrency or parallelism in system.

Serializability identifies those executions of transactions guaranteed to ensure consistency.

The sheduler



The scheduler component of a DBMS must ensure that the individual steps of different transactions preserve consistency.

Serializability - some definitions

Schedule: time-ordered sequence of the important actions taken by one or more transitions.

Serial Schedule: a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

No guarantee that results of all serial executions of a given set of transactions will be identical.

Serializability - some definitions

Nonserial Schedule: Schedule where operations from a set of concurrent transactions are interleaved.

The objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.

In other words, want to find nonserial schedules that are equivalent to some serial schedule. Such a schedule is called serializable.

Serializability - some important rules

In serializability, ordering of read/writes is important:

(a) If two transactions only read a data item, they do not conflict and order is not important.

(b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.

(c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Example of Conflict Serializability

Time	T_7	T_8	T_7	T_8	T_7	T_8
t_1	begin_transaction		begin_transaction		begin_transaction	
t_2	read(bal_x)		read(bal_x)		read(bal_x)	
t_3	write(bal_x)		write(bal_x)		write(bal_x)	
t_4		begin_transaction		begin_transaction	read(bal_y)	
t_5		read(bal_x)		read(bal_x)	write(bal_y)	
t_6		write(bal_x)	read(bal_y)		commit	
t_7	read(bal_y)			write(bal_x)		begin_transaction
t_8	write(bal_y)		write(bal_y)			read(bal_x)
t_9	commit		commit			write(bal_x)
t_{10}		read(bal_y)		read(bal_y)		read(bal_y)
t_{11}		write(bal_y)		write(bal_y)		write(bal_y)
t_{12}		commit		commit		commit
	(a)		(b)		(c)	

Serializability

Conflict serializable schedule orders any conflicting operations in same way as some serial execution.

Constrained write rule: transaction updates data item based on its old value, which is first read.

Under the constrained write rule we can use precedence graph to test for serializability.

Precedence Graph

Given a schedule S, a precedence graph is a directed graph G= (N,E) where

- N = set of nodes
- E = set of directed edges

Created as follows:

- create a node for each transaction;
- a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i;
- a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.

An example

The precedence graph for the schedule (a) is:

Precedence Graph and serializability

If an edge $T_i \rightarrow T_j$ exists in the precedence graph for S, then in any serial schedule S' equivalent to S, T_i must appear before T_j

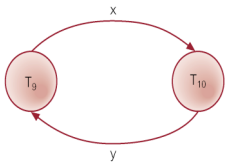
If the precedence graph contains cycle schedule is not conflict serializable.

Example - Non-conflict serializable schedule

- T9 is transferring £100 from one account with balance bal_x to another account with balance bal_y.
- T10 is increasing balance of these two accounts by 10%.

Example - Non-conflict serializable schedule

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	



View Serializability

Offers less stringent definition of schedule equivalence than conflict serializability.

Two schedules S_1 and S_2 are view equivalent if:

- For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.
- For each read on x by T_i in S_1, if value read by x is written by T_j, T_i must also read value of x produced by T_j in S_2.
- For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.

View Serializability

- Schedule is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable, although converse is not true.
- It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.
- In general, testing whether schedule is serializable is NP-complete.

Example - View Serializable schedule

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6	write(bal_x)		
t_7	commit		
t_8			begin_transaction
t_9			write(bal_x)
t_{10}			commit

Recoverability

Serializability identifies schedules that maintain database consistency, assuming no transaction fails.

Could also examine recoverability of transactions within schedule.

If transaction fails, atomicity requires effects of transaction to be undone.

Durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction).

Recoverable Schedule

A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

How can the DBMS ensure serializability?

Concurrency Control Techniques

Two basic concurrency control techniques:

- Locking methods
- Timestamping

Both are conservative approaches: delay transactions in case they conflict with other transactions.

Optimistic methods assume conflict is rare and only check for conflicts at commit.

Locking

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

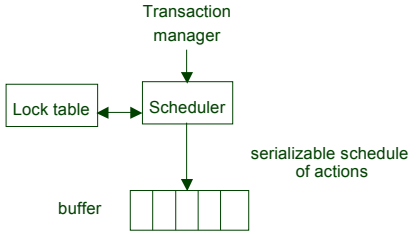
Generally, a transaction must claim a

- read (shared), or
- write (exclusive)

lock on a data item before read or write.

Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking



Locking - Basic Rules

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.
- Some systems allow transaction to upgrade a shared lock to an exclusive lock, or vice-versa.

Example - Incorrect Locking Schedule

Time	T_9	T_{10}
t1	begin_transaction	
t2	read(bal_x)	
t3	$bal_x = bal_x + 100$	
t4	write(bal_x)	begin_transaction
t5		read(bal_x)
t6		$bal_x = bal_x * 1.1$
t7		write(bal_x)
t8		read(bal_y)
t9		$bal_y = bal_y * 1.1$
t10		write(bal_y)
t11	read(bal_y)	commit
t12	$bal_y = bal_y - 100$	
t13	write(bal_y)	
t14	commmit	

Example - Incorrect Locking Schedule

A valid schedule using the basic rules of locking is:

S = {write_lock(T_9, bal_x),
read(T_9, bal_x), write(T_9, bal_x),
unlock(T_9, bal_x),
write_lock(T_{10}, bal_x),
read(T_{10}, bal_x), write(T_{10}, bal_x),
unlock(T_{10}, bal_x),
write_lock(T_{10}, bal_y),
read(T_{10}, bal_y), write(T_{10}, bal_y),
unlock(T_{10}, bal_y),
commit(T_{10}),
write_lock(T_9, bal_y),
read(T_9, bal_y), write(T_9, bal_y),
unlock(T_9, bal_y),
commit(T_9) }

Example - Incorrect Locking Schedule

If at start, bal_x = 100, bal_y = 400, result should be:

- bal_x = 220, bal_y = 330, if T_9 executes before T_{10}, or
- bal_x = 210, bal_y = 340, if T_{10} executes before T_9 .

However, result gives bal_x = 220 and bal_y = 340.

S is not a serializable schedule.

Example - Incorrect Locking Schedule

- Problems arise when transactions release locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.

Two-Phase Locking (2PL)

A transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

We can split the transaction in two phases:

- Growing phase - acquires all locks but cannot release any locks.
- Shrinking phase - releases locks but cannot acquire any new locks.

Preventing Lost Update Problem using 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing Uncommitted Dependency Problem using 2PL

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing Inconsistent Analysis Problem using 2PL

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Cascading Rollback

If every transaction in a schedule follows 2PL, schedule is serializable.

However, problems can occur with interpretation of when locks can be released.

Cascading Rollback - an example

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(bal_x)		
t_3	read(bal_x)		
t_4	read_lock(bal_y)		
t_5	read(bal_y)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(bal_x)		
t_8	unlock(bal_x)	begin_transaction	
t_9	⋮	write_lock(bal_x)	
t_{10}	⋮	read(bal_x)	
t_{11}	⋮	$bal_x = bal_x + 100$	
t_{12}	⋮	write(bal_x)	
t_{13}	⋮	unlock(bal_x)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock(bal_x)
t_{18}		rollback	⋮
t_{19}			rollback

Cascading Rollback

Transactions conform to 2PL, but …

- T14 aborts.
- Since T15 is dependent on T14, T15 must also be rolled back. Since T16 is dependent on T15, it too must be rolled back. Cascading rollback.
- To prevent this with 2PL, leave release of all locks until end of transaction.

Locking methods: problems

Deadlock: An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Deadlock - an example

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock - possible solutions?

Only one way to break deadlock: abort one or more of the transactions.

Deadlock should be transparent to user, so DBMS should restart transaction(s).

Two general techniques for handling deadlock:

- Deadlock prevention.
- Deadlock detection and recovery.

Timeouts

Transaction that requests lock will only wait for a system-defined period of time.

If lock has not been granted within this period, lock request times out.

In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

Deadlock Prevention

DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.

Could order transactions using transaction timestamps:

- Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (dies) and restarted with same timestamp.
- Wound-Wait - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).

Deadlock Detection and Recovery

DBMS allows deadlock to occur but recognizes it and breaks it.

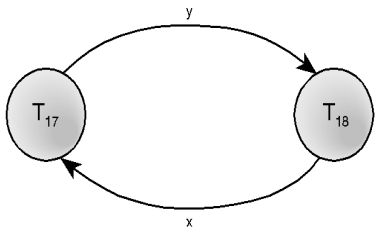
Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:

- Create a node for each transaction.
- Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.

Deadlock exists if and only if WFG contains cycle.

WFG is created at regular intervals.

Example - Wait-For-Graph (WFG)



Recovery from Deadlock Detection

Several issues:

- choice of deadlock victim;
- how far to roll a transaction back;
- avoiding starvation.

Timestamping

- Transactions ordered globally so that older transactions, transactions with earlier timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

Timestamping

Timestamp: a unique identifier created by DBMS that indicates relative starting time of a transaction.

Can be generated by:

- using system clock at time transaction started, or
- incrementing a logical counter every time a new transaction starts.

Timestamping - definition

Timestamping: a concurrency control protocol that orders transactions in such a way that older transactions get priority in the event of a conflict.

Read/write proceeds only if last update on that data item was carried out by an older transaction.

Otherwise, transaction requesting read/write is restarted and given a new timestamp.

Also timestamps for data items:

- read-timestamp: timestamp of last transaction to read item.
- write-timestamp: timestamp of last transaction to write item.

Timestamping: how does the protocol work?

A transaction T with timestamp ts(T) wants to read(x):

- ts(T) < write_timestamp(x)
  - x already updated by younger (later) transaction.
  - Transaction T must be aborted and restarted with a new timestamp.
- ts(T) ≥ write_timestamp(x)
  - transaction can proceed
  - read_timestamp = max(ts(T), read_timestamp(x))

Timestamping: how does the protocol work?

A transaction T with timestamp ts(T) wants to write(x):

- ts(T) < read_timestamp (x)
  - younger transaction has read the value x
  - rollback transaction T and restart using a later timestamp
- ts(T) < write_timestamp(x)
  - x already written by younger transaction.
  - Write can safely be ignored - ignore obsolete write rule.
- all other cases: operation accepted and executed.

Example

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)+	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

+ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.
‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Rare Conflicts

In some systems we can safely assume that conflicts are rare.

Do we really need locking or timestamping protocols?

No. More efficient techniques can be used to let transactions proceed without delays to ensure serializability.

Optimistic Techniques

- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.
- Three phases:
  - Read
  - Validation
  - Write.

Optimistic Techniques - Read Phase

- Extends from start until immediately before commit.
- Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.

Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

Optimistic Techniques - Write Phase

- Follows successful validation phase for update transactions.
- Updates made to local copy are applied to the database.

Granularity of Data Items

Size of data items chosen as unit of protection by concurrency control protocol.

Ranging from coarse to fine:

- The entire database.
- A file.
- A page (or area or database spaced).
- A record.
- A field value of a record.

Granularity of Data Items

Tradeoff:

- coarser, the lower the degree of concurrency.
- finer, more locking information that is needed to be stored.

Best item size depends on the types of transactions!!

Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.

Hierarchy of Granularity

- Intention lock could be used to lock all ancestors of a locked node.
- Intention locks can be read or write. Applied top-down, released bottom up.

Table 20.1 Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible

Levels of Locking