Transactions

SET08104 Database Systems

Copyright @ Napier University

Concurrency using Transactions

The goal in a ‘concurrent’ DBMS is to allow multiple users to access the database simultaneously without interfering with each other.

A problem with multiple users using the DBMS is that it may be possible for two users to try and change data in the database simultaneously. If this type of action is not carefully controlled, inconsistencies are possible.

To control data access, we first need a concept to allow us to encapsulate database accesses. Such encapsulation is called a ‘Transaction’.

Transactions

- Transaction (ACID)
  - unit of logical work and recovery
    - A - atomicity (for integrity)
    - C - consistency preservation
    - I - isolation
    - D - durability
- Available in SQL
- Some applications require nested or long transactions

Transactions cont...

After work is performed in a transaction, two outcomes are possible:

- Commit - Any changes made during the transaction by this transaction are committed to the database.
- Abort - All the changes made during the transaction by this transaction are not made to the database. The result of this is as if the transaction was never started.

Transaction Schedules

A transaction schedule is a tabular representation of several transactions executed over time. This is useful when examining problem scenarios. Within the diagrams various nomenclatures are used:

- READ(*a*) - a read action on an attribute or data item called ‘a’.
- WRITE(x,*a*) - a write action on an attribute or data item called ‘a’, where the value ‘x’ is written into ‘a’.
- tn (e.g. t1,t2,t10) - indicates the time at which something occurred. The units are not important, but t*n* always occurs before t*n+1*.

Schedules cont...

Consider transaction A, which loads a bank account balance X (initially 20) and adds 10 pounds to it. Such a schedule would look like this:

Time	Transaction A
t1	TOTAL:=READ(X)
t2	TOTAL:=TOTAL+10
t3	WRITE(TOTAL,X)

Schedules cont...

Now consider that, at the same time as trans A runs, trans B runs. Transaction B gives all accounts a 10% increase. Will X be 32 or 33?

Schedules cont...

Time	Transaction A	Value TOTAL	Transaction B	Value BALANCE
t1			BONUS:=READ(X)	20
t2	TOTAL:=READ(X)	20		
t3	TOTAL:=TOTAL+10	30		
t4	WRITE(TOTAL,X)	30		
t5			BONUS:=BONUS*110%	22
t6			WRITE(BONUS,X)	22

Woops... X is 22! Depending on the interleaving, X can also be 32, 33, or 30. Lets classify erroneous scenarios.

Lost Update scenario

Time	Transaction A	Transaction B
t1	X = READ(R)	
t2		Y = READ(R)
t3	WRITE(X,R)	
t4		WRITE(Y,R)

Transaction A's update is lost at t4, because Transaction B overwrites it. B missed A's update at t4 as it got the value of R at t2.

Uncommitted Dependency

Time	Transaction A	Transaction B
t1		WRITE(X,R)
t2	Y = READ(R)	
t3		**ABORT**

Transaction A is allowed to READ (or WRITE) item R which has been updated by another transaction but not committed (and in this case ABORTed).

Inconsistency Scenario

Time	X	Y	Z	Transaction A		Transaction B
				Action	SUM	
t1	40	50	30	SUM:=READ(X)	40	
t2	40	50	30	SUM+=READ(Y)	90	
t3	40	50	30			ACC1 = READ(Z)
t4	40	50	20			WRITE(ACC1-10,Z)
t5	40	50	20			ACC2 = READ(X)
t6	50	50	20			WRITE(ACC2+10,X)
t7	50	50	20			COMMIT
t8	50	50	20	SUM+=READ(Z)	110	
				SUM should have been 120		

Serializability

- A ‘schedule’ is the actual execution sequence of two or more concurrent transactions.
- A schedule of two transactions T1 and T2 is ‘serializable’ if and only if executing this schedule has the same effect as either T1;T2 or T2;T1.

Precedence Graph

In order to know that a particular transaction schedule can be serialized, we can draw a precedence graph. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions.

The schedule is said to be serialised if and only if there are no cycles in the resulting diagram.

Precedence Graph : Method

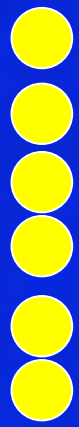
To draw one;

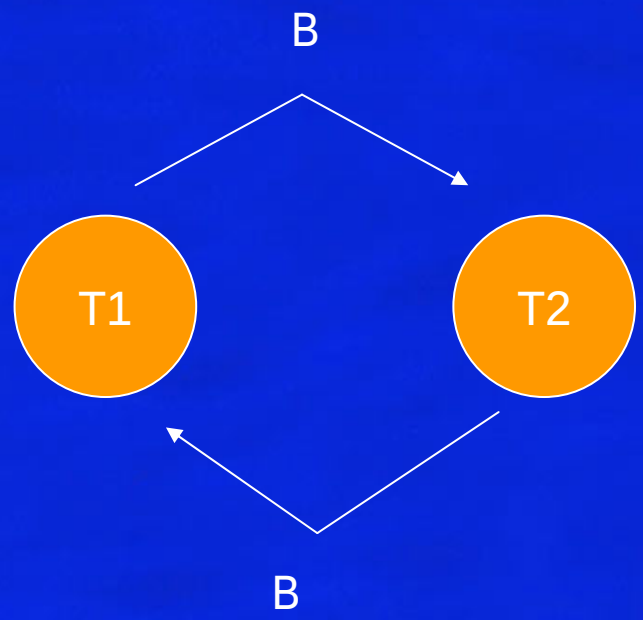
1. Draw a node for each transaction in the schedule
2. Where transaction T1 writes to an attribute which transaction T2 has read from, draw a line pointing from T2 to T1.
3. Where transaction T1 writes to an attribute which transaction T2 has written to, draw a line pointing from T2 to T1.
4. Where transaction T1 reads from an attribute which transaction T2 has written to, draw a line pointing from T2 to T1.

Example 1

Consider the following Schedule:

Time	T1	T2
t1	READ(A)	
t2	READ(B)	
t3		READ(A)
t4		READ(B)
t5	WRITE(x,B)	
t6		WRITE(y,B)

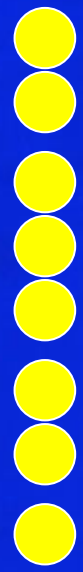


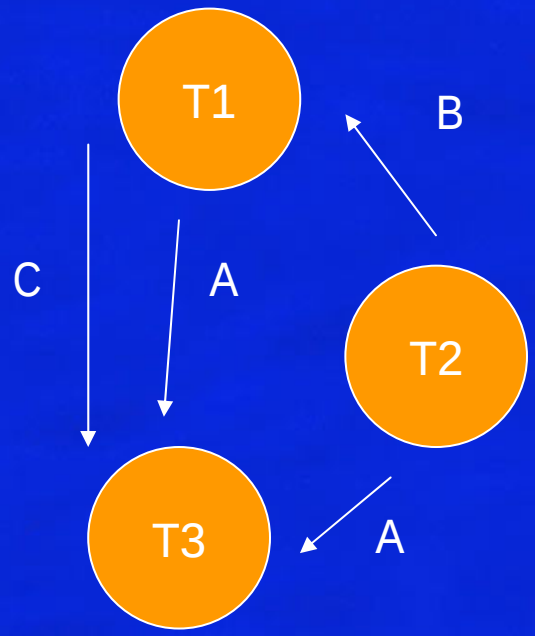


Example 2

Consider the following Schedule:

Time	T1	T2	T3
t1	READ(A)		
t2	READ(B)		
t3		READ(A)	
t4		READ(B)	
t5			WRITE(x,A)
t6	WRITE(v,C)		
t7	WRITE(w,B)		
t8			WRITE(z,C)





Locking

A solution to enforcing serialisability?

- read (shareable) lock
- write (exclusive) lock
- coarse granularity
  - easier processing
  - less concurrency
- fine granularity
  - more processing
  - higher concurrency

Locking cont...

Many systems use locking mechanisms for concurrency control. When a transaction needs an assurance that some object will not change in some unpredictable manner, it acquires a lock on that object.

- A transaction holding a read lock is permitted to read an object but not to change it.
- More than one transaction can hold a read lock for the same object.
- Usually, only one transaction may hold a write lock on an object.
- On a transaction schedule, we use ‘S’ to indicate a shared lock, and ‘X’ for an exclusive write lock.

Locking – Uncommitted Dependency

Locking solves the uncommitted dependency problem

Time	Transaction A	Transaction B	Lock on R Before => after
t1		write(p,R)	- => X
t2	read(R) **WAIT**		
t3	…**WAIT…**	**ABORT**	X => -
t4	read(R) **GO**		- -> S

Deadlock

Deadlock can arise when locks are used, and causes all related transactions to WAIT forever

Time	Transaction A	Transaction B	Lock State	
			X	**Y**
t1	write(p,X)		- => X	-
t2		write(q,Y)	X	- => X
t3	read(Y) **WAIT**		X	X
t4	…**WAIT…**	read(X) **WAIT**	X	X
t5	…**WAIT…**	…**WAIT…**	X	X

Deadlock cont…

The `lost update' senario results in deadlock with locks. So does the `inconsistency' scenario.

Time	Transaction A	Transaction B	Lock on R Before => after
t1	read(R)		- => S
t2		read(R)	S => S
t3	write(p,R)		S
t4	…**WAIT**…	write(q,R)	S
t5	…**WAIT**…	…**WAIT**…	S

Deadlock Handling

- Deadlock avoidance
  - pre-claim strategy used in operating systems
  - not effective in database environments.
- Deadlock detection
  - whenever a lock requests a wait, or on some perodic basis.
  - if a transaction is blocked due to another transaction, make sure that the transaction is not blocked on the first transaction, either directly or indirectly via another transaction.

Deadlock Resolution

If a set of transactions is considered to be deadlocked:

1. choose a victim (e.g. the shortest-lived transaction)
2. rollback ‘victim’ transaction and restart it.
   - The rollback terminates the transaction, undoing all its updates and releasing all of its locks.
   - A message is passed to the victim and depending on the system the transaction may or may not be started again automatically.

Two-Phase Locking

The presence of locks does not guarantee serialisability. If a transaction is allowed to release locks before the transaction has completed, and is also allowed to acquire more (or even the same) locks later then the benefit of locking is lost.

If all transactions obey the ‘two-phase locking protocol’, then all possible interleaved executions are guaranteed serialisable.

Two-Phase locking cont...

The two-phase locking protocol:

- Before operating on any item, a transaction must acquire at least a shared lock on that item. Thus no item can be accessed without first obtaining the correct lock.
- After releasing a lock, a transaction must never go on to acquire any more locks.

The technical names for the two phases of the locking protocol are the ‘lock-acquisition phase’ and the ‘lock-release phase’.

Other Database Consistency Methods

Two-phase locking is not the only approach to enforcing database consistency. Another method used in some DMBS is timestamping. With timestamping, there are no locks to prevent transactions seeing uncommitted changes, and all physical updates are deferred to commit time.

- locking synchronises the interleaved execution of a set of transactions in such a way that it is equivalent to some serial execution of those transactions.
- timestamping synchronises that interleaved execution in such a way that it is equivalent to a particular serial order - the order of the timestamps.

Timestamping rules

The following rules are checked when transaction T attempts to change a data item. If the rule indicates ABORT, then transaction T is rolled back and aborted (and perhaps restarted).

- If T attempts to read a data item which has already been written to by a younger transaction then ABORT T.
- If T attempts to write a data item which has been read from or written to by a younger transaction then ABORT T.

If transaction T aborts, then all other transactions which have seen a data item written to by T must also abort. In addition, other aborting transactions can cause further aborts on other transactions. This is a ‘cascading rollback’.