Database Systems 2

Lecture 15

Transactions and Concurrency

Concurrency using Transactions

The goal in a concurrent DBMS is to allow multiple users to access the database simultaneously without them being aware of each other.

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

To control data access, we first need a concept which allows us to encapsulate database accesses.

Transaction is the name given to this concept.

Transactions

Transaction - unit of logical work and recovery

Desirable properties of a transaction:- ACID

- atomicity (for integrity)
- consistency preservation
- isolation
- durability

Transaction handling is available in SQL.

Some applications require nested or long transactions

Transactions cont...

To help handle a transaction, SQL provides three commands:

COMMIT

Any changes made during the transaction by this transaction are committed to the database.

ROLLBACK

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.

Most commands (Alter, Insert, Create, Select) implicitly open a transaction. However, the user can also do this explicitly.

SET TRANSACTION

Allows you start a new transaction and specify parameters such as Read Only, Read Write, Isolation Level. (aka START TRANSACTION)

Example in Oracle

```
SET TRANSACTION NAME 'Transfer funds';

UPDATE cashtable
SET balance = balance - 200
WHERE customerID = 'BPCCollege';

UPDATE stafftable
SET wages = wages + 200
WHERE staffid = 'Wilson';

COMMIT;
```

See http://docs.oracle.com/cd/B19306_01/appdev.102/b14261/settransaction_statement.htm

Transaction Schedules

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

READ(*a*)

- This is a read action on an attribute or data item called ‘a’.

WRITE(*a*)

- This is a write action on an attribute or data item called ‘a’.

WRITE(*a*) ← x

- This is 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)

- This indicates the time at which something occurred. The units are not important, but t*n* always occurs before t*n+1*.

Schedules cont...

So if we wanted the show the transaction from slide 5 in this format:

Time	Transaction 'Transfer Funds'
t1	amount1 ← READ(cashtable.balance)
t2	amount1 ← amount1 – 200
t3	WRITE(cashtable.balance) ← amount1
t4	amount2 ← READ(stafftable.wages)
t5	amount2 ← amount2 + 200
t6	WRITE(stafftable.wages) ← amount2
t7	COMMIT

Schedules cont...

Now consider that, at the same time as trans A runs, trans B runs.

Either the computer has two separate processors, so they can run in parallel – or it has to timeshare, and interleave the two transactions.

Time	Transaction A	Transaction B
t1	total1 ← READ(X)	
t2	total1 ← total1 + 10	balance ← READ(X)
t3	WRITE(X) ← total1	balance ← balance * 110%
t4		WRITE(X) ← balance
t5		
t6		

Whoops… X is 22! Depending on the interleaving, X can also be 32, 33, or 30. Let's classify error scenarios.

Lost Update Problem

Time	Transaction A	Transaction B	x
t1		*Begin_trans*	
t2	*Begin_trans*	b ← READ(x)	
t3	a← READ(x)	b ← b + 100	
t4	a ← a – 10	WRITE(x) ← b	
t5	WRITE(x) ← a	*Commit*	
t6	*Commit*		

Transaction B's update is lost at t5, because Transaction A overwrites it. A missed B's update at t5 as it got the value of X at t3.

Uncommitted Dependency

Time	Transaction A	Transaction B	x
t1		*Begin_trans*	
t2		b ← READ(x)	
t3		b ← b + 100	
t4	*Begin_trans*	WRITE(x) ← b	
t5	a ← READ(x)		
t6	a ← a – 10	***Rollback***	
t7	WRITE(x) ← a		
t8	*Commit*		

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

Inconsistent Analysis Scenario

The above two problems relate to transactions that are updating the database.

Problems can also occur when transactions are just reading the database.

This is know as a dirty read or unrepeatable read.

Inconsistent analysis occurs when a transaction reads several values from a database, but a second transaction updates some of them during execution of the first.

Inconsistent Analysis Scenario

Time	Transaction A	Transaction B	X	Y	Z	sum
t1		*Begin_trans*				
t2	*Begin_trans*	sum ← 0				
t3	a ← READ(x)	c ← READ(x)				
t4	a ← a – 10	sum ← sum + c				
t5	WRITE(x) ← a	d ← READ(y)				
t6	b ← READ(z)	sum ← sum + d				
t7	b ← b + 10					
t8	WRITE(z) ← b					
t9	*Commit*	e ← READ(z)				
t10		sum ← sum + e				

The value of sum contains a value based on the old value of x, but on the new value of z. This means that Transaction B is not isolated from Transaction A.

Serialisability

The objective of concurrency control is to avoid the problems described above.

The obvious way to acheive this is to only allow one transaction to begin after the preceding one has committed.

However, a multi-user database system aims to maximise the degree of concurrency.

A ‘schedule’ is the actual execution sequence of two or more concurrent transactions.

A schedule of two transactions T1 and T2 is ‘serialisable’ if and only if executing this schedule has the same effect as either:

T1 followed by T2 or T2 followed by T1.

If it is serialisable, it means that the operations from the transactions can be interleaved.

Precedence Graph

In order to know that a particular transaction schedule can be serialised, 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 (or serialisable) if and only if there are no cycles (closed loops) in the resulting diagram.

Precedence Graph : Method

To draw one;

Draw a node for each transaction in the schedule

Where transaction A writes to an attribute which transaction B has read from, draw an line pointing from B to A.

Where transaction A writes to an attribute which transaction B has written to, draw a line pointing from B to A.

Where transaction A reads from an attribute which transaction B has written to, draw a line pointing from B to A.

To summarise….

If

Trans A		Trans B	
Writes to att	when	has read from att	or
Writes to att	when	has written to att	or
Reads from att	when	has written to att	

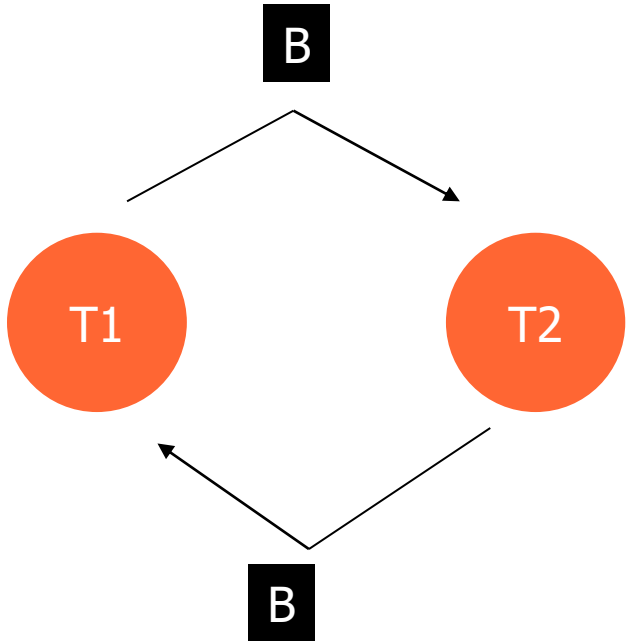
Then

Draw arrow from Trans B to Trans A

Example 1

Consider the following Schedule:

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

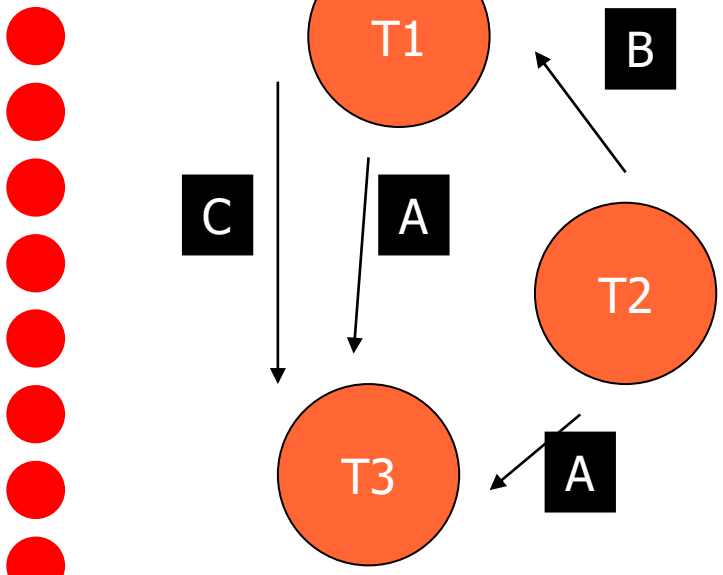


Not Serialisable

Example 2

Consider the following Schedule:

Time	T1	T2	T3
t1	READ(A)		
t2	READ(B)		
t3		READ(A)	
t4		READ(B)	
t5			WRITE(A)
t6	WRITE(C)		
t7	WRITE(B)		
t8			WRITE(C)



Serialisable

Locking

A solution to enforcing serialisability.

- read (shareable) lock
- write (exclusive) lock
- coarse granularity (Tables)
  - easier processing
  - less concurrency
- fine granularity (Rows or data items)
  - 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.
- A transaction holding a write lock can read and update the item.
- Only one transaction may hold a write lock on an object. No other transaction may even read that item.

Locking – Uncommitted Dependency

Locking solves the uncommitted dependency problem

Time	Transaction A	Transaction B
t1		Request_Write_lock(R)
t2	Request_Read_lock(R)	Grant_Write_lock(R)
t3	*wait*	WRITE(R)
t4	*wait*	*wait*
t5	*wait*	ROLLBACK/UNLOCK(R)
t6	Grant_Read_lock(R)	
t7	READ(R)	

The other problems are solved in a similar fashion.

However, there is one problem that can arise with the use of locking.

Deadlock

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

Time	Transaction A	Transaction B
t1	Request_write_lock(X)	
t2	WRITE(X)	
t3	*wait*	Request_write_lock(Y)
t4	*wait*	WRITE(Y)
t5	Request_read_lock(Y)	*wait*
t6	*wait*	Request_read_lock(X)
t7	*wait*	*wait*

Deadlock Resolution

If a set of transactions is considered to be deadlocked:

- Choose a victim (e.g. the shortest-lived transaction)
- 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 alone 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 (Read) 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’.