Concurrency

Database Systems

Michael Pound

This Lecture

- Concurrency control
- Serialisability
  - Schedules of transactions
  - Serial and serialisable schedules
- Locks
- 2 Phase Locking Protocol
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22

Transactions so Far

- Transactions are the 'logical unit of work' in a database
  - ACID properties
  - Also the unit of recovery
- Transactions will involve some read and/or writes on a database
- COMMIT
  - Signals the successful end of a transaction
  - Changes are made permanent and visible to other transactions
- ROLLBACK
  - Signals the unsuccessful end of a transaction
  - Changes are undone

Transactions so Far

- Atomicity
  - Transactions conceptually have no component parts
  - The run completely, or not at all
- Consistency
  - Transactions take the database from one consistent state to another
- Isolation
  - Incomplete transactions are invisible to others until they have committed
- Durability
  - Committed transactions must be made permanent

Concurrency

- Large databases are used by many people
  - Many transactions are to be run on the database
  - It is helpful to run these simultaneously
  - Still need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Easy to preserve atomicity and isolation
  - Long transactions (e.g. backups) will delay others

Concurrency Problems

- In order to run two or more concurrent transactions, their operations must be interleaved
- Each transaction gets a share of the computing time
- This can lead to several problems
  - Lost updates
  - Uncommitted updates
  - Incorrect updates
- All arise when 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 both read X, both modify it, then both write it out
  - The net effect of both transactions should be no change to X
  - Only T2's change is seen however

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 then rolled back
  - The change made by T1 is rolled back
  - 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 records a change
  - T1 consists of two parts - take 5 from X then add 5 to Y
  - T2 sees the effect of the first change, but not the second

Concurrency Control

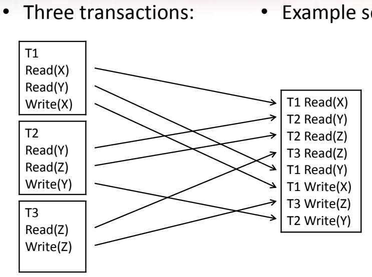
- Concurrency control is the process of managing simultaneous operations on the database without having them interfere with each other
  - Possibly reading and writing the same data
  - Long transactions must not hold up others
  - ACID properties must be maintained

Schedules

- A *schedule* is a sequence of the operations in a set of concurrent transactions that preserves the order of operations in each of the individual transactions
- A *serial* schedule is a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions (each must commit before the next can begin)

Example Schedule

- Three transactions:
- Example schedule



Example Schedule

- Three transactions:

T1
Read(X)
Read(Y)
Write(X)

T2
Read(Y)
Read(Z)
Write(Y)

T3
Read(Z)
Write(Z)

- Example serial schedule

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

Serial Schedules

- A serial schedule is guaranteed to avoid interference between transactions, and preserve database consistency
- However, this approach does not allow for concurrent access, i.e. Interleaving operations from multiple transactions

Serialisability

- Two schedules are equivalent if they always have the same effect
- A schedule is *serialisable* if it is equivalent to some serial schedule
- For example:
  - If two transactions only read from some data items, the order in which they do this is not important
  - If T1 reads and then updates X, and T2 reads then updates Y, then again this can occur in any order

Serial and Serialisable

- Interleaved (nonserial) Schedule

T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)

This schedule is serialisable – has the same effect as a serial schedule

- Serial Schedule

T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)

Conflict Serialisability

- Two transactions have a confict:
  - NO If they refer to different resources
  - NO If they only read
  - YES If at least one is a write and they use the same resource
- A schedule is conflict serialisable if the transactions in the schedule have a conflict, but the schedule is still serialisable

Conflict Serialisable Schedule

- Interleaved Schedule

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

This schedule is serialisable, even though T1 and T2 read and write the same resources X and Y: They have a conflict

- Serial Schedule

T1 Read(X)
T1 Write(X)
T1 Read(Y)
T1 Write(Y)

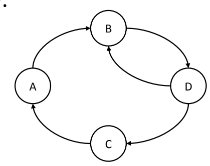
T2 Read(X)
T2 Write(X)
T2 Read(Y)
T2 Write(Y)

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
  - How do we determine whether or not a schedule is conflict serialisable?
  - How do we construct conflict serialisable schedules

Graphs

- In mathematics, a graph is a structure (V,E) of Vertices and Edges. In the case of a directed graph, these edges include directions
- For example:



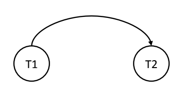
Precedence Graphs

- To determine if a schedule is conflict serialisable we use a precedence graph
  - Transactions are vertices of the graph
  - There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serialisable schedule
- Edge T1 → T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R)
  - T1 Write(R) followed by T2 Read(R)
  - T1 Write(R) followed by T2 Write(R)
- The schedule is serialisable if there are no cycles

Precedent Graph Example

- No cycles: conflict serialisable schedule

- T1 reads X before T2 writes X
- T1 writes X before T2 reads X
- T1 writes X before T2 writes X

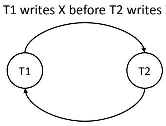


T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

Precedent Graph Example

- The lost update problem has this precedence graph:

- T1 reads X before T2 writes X
- T1 writes X before T2 writes X



- T2 reads X before T1 writes X

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

Locking

- Locking is a procedure used to control concurrent access to data (to ensure serialisability of concurrent transactions)
- In order to use a ‘resource’ a transaction must first acquire a lock on that resource
  - A resource could be a single item of data, some or all of table, or even a whole database
- This may deny access to other transactions to prevent incorrect results

Lock Types

- There are two types of lock
  - Shared lock (often called a read lock)
  - Exclusive lock (often called a write lock)
- Read locks allow several transactions to read data simultaneously, but none can write to that data
- Write locks allow a single transaction exclusive access to read and write a resource

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
- A lock might be released during execution when no longer needed, or upon COMMIT or ROLLBACK

Locking

- A transaction may not acquire a lock on any resource that is currently write-locked by another transaction
- A transaction may not acquire a write lock on any resource that is currently locked by another transaction
- If the requested lock is not available, the transaction waits
- The DBMS is responsible for issuing locks

Locking Example

- Locking won't successfully allow us to serialise all schedules. For example:

	T1	T2	
write-lock(X)	Read(X)		
	X = X - 500		
unlock(X)	Write(X)		
		Read(X)	write-lock(X) write-lock(Y)
		Read(Y)	
		X = X * 1.1	
		Y = Y * 1.1	
		Write(X)	
		Write(Y)	unlock(X) unlock(Y)
write-lock(Y)	Read(Y)		
	Y = Y + 500		
unlock(Y)	Write(Y)		

Two-Phase Locking

- A transaction follows two-phase locking protocol (2PL) if all locking operations precede all unlocking operations
- Other operations can happen at any time throughout the transaction
- Two phases:
  - **Growing** phase where locks are acquired
  - **Shrinking** phase where locks are released

Two-Phase Locking Example

- T1 follows 2PL protocol
  - All locks in T1 are acquired before any are released
  - This happens even if the resource is no longer used
- T2 does not
  - Releases a lock on X, which is no longer needed, before acquiring on Y

T1	T2
write-lock(X)	write-lock(X)
Read(X)	Read(X)
X = X + 100	X = X + 100
Write(X)	Write(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y – 100	Y = Y – 100
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)
COMMIT	COMMIT

Serialisability Theorem

Any schedule of two-phase locking transactions is conflict serialisable

2PL Prevents Lost Update

	T1	T2	
write-lock(X)	Read(X)		
	X = X – 5		write-lock(X)
		WAIT	
	Write(X)	WAIT	
unlock(X)	COMMIT	WAIT	
		Read(X)	
		X = X + 5	
		Write(X)	
		COMMIT	unlock(X)

2PL Prevents Uncommitted Update

	T1	T2	
write-lock(X)	Read(X)		
	X = X – 5		write-lock(X)
	Write(X)	WAIT	
unlock(X)	ROLLBACK	WAIT	
		WAIT	
		Read(X)	
		X = X + 5	
		Write(X)	
		COMMIT	unlock(X)

The value of X is restored during rollback, before the write-lock is released

2PL Prevents Inconsistent Analysis

	T1	T2	
write-lock(X)	Read(X)		
	X = X – 5		
	Write(X)		read-lock(X)
		WAIT	
write-lock(Y)	Read(Y)	WAIT	
	Y = Y + 5	WAIT	
	Write(Y)	WAIT	
		WAIT	
unlock(X) unlock(Y)	COMMIT	WAIT	
		Read(X)	
		Read(Y)	
		Sum = X + Y	
		...	

Concurrency in SQL

- Concurrency in MySQL (and most other DBMSs) is handled automatically
- UPDATE, INSERT, DELETE etc will obtain write locks
- SELECT will obtain a read lock – or may read an old cached value
- In MySQL, Locking protocol depends on the engine
  - MyISAM: Table Level Locking
  - Memory: Table Level Locking
  - InnoDB: Row Level Locking

Concurrency in MySQL

- Sometimes you might want to lock a resource specifically for updating:

```
SELECT ID FROM Artist WHERE Name =
  'Muse';
... Some processing
INSERT INTO CD VALUES (NULL, 2, 'The
  Resistance', 9.99, 'Rock');
```

- In the short time between these queries, the ID for muse may have been written to

Locking in a SELECT

- For times when a Subquery isn't appropriate:

```
SELECT *
 FROM table
 WHERE ...
 FOR UPDATE;
```

- `FOR UPDATE` write-locks all rows that we read until the end of the transaction.
- It has the added benefit of reading the very latest values of these rows (not using cached values)
- You can use `LOCK IN SHARE MODE` to obtain a read lock instead

This Lecture in Exams

Define the term *Schedule*, *Serial Schedule* and *Serialisable* in the context of database concurrency

Explain the Lost Update problem, and provide an example schedule demonstrating this

Explain how two-phased locking protocol can avoid the lost update problem

Next Lecture

- Deadlocks
  - Deadlock detection
  - Deadlock prevention
- Timestamping
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22