COL362/632 Introduction to Database Management Systems

Concurrency Control

Kaustubh Beedkar

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Concurrent Transactions

- Improved throughput & resource utilization
- Low latency
- But, **how do we enforce serializability?**

Enforcing Serializability

Broadly two approaches:

1. **Pessimistic Approach**
   - Use locks
   - Two-phase Locking (2PL)
   - Timestamp-based Protocols

2. **Optimistic Approach**
   - Let transactions execute; deal with conflicts later
   - Optimistic Concurrency Control (OCC)
   - Multi-version Concurrency Control (MVCC)

Locks

- Mechanism to control concurrent access

- Two modes
  1. **Shared mode (lock-s):**
     Data item can only be read
  2. **Exclusive mode (lock-x):**
     Data item can be both read and written

- Concurrency Manager manages lock requests
  - Grants/ denies lock requests

Lock Compatibility Matrix

	lock-s	**lock-x**
lock-s	compatible	non-compatible
lock-x	non-compatible	non-compatible

- Any number of transactions can hold shared locks on a data item
- But, if a transaction holds an exclusive lock on the item, then no other transaction can hold any lock on the item

Locking Example

Consider the following schedule

T_1
lock-s(A)
read(A)
unlock(A)
lock-s(B)
read(B)
unlock(B)
print(A+B)

T_1	T_2
lock-s(A)	
read(A)	
unlock(A)	
	lock-x(A)
	lock-x(B)
	read(A); write(A)
	read(B); write(B)

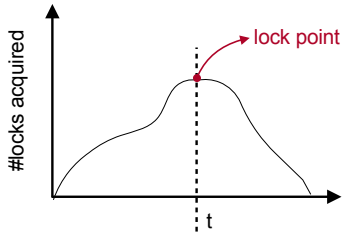
- Simply locking does not guarantee serializability!
- Must follow a set of rules

Two Phase Locking (2PL)

- A Tx must acquire a lock-s before reading
- A Tx must acquire a lock-x before writing
- A Tx cannot get any new locks after releasing a lock

Two phases

1. Growing phase
   - A Tx may acquire locks
   - A Tx cannot release locks
2. Shrinking phase
   - A Tx may release locks
   - A Tx cannot acquire locks



- **2PL guarantees conflict serializability**

  Proof sketch: order of lock points gives us equivalent serial schedule

Two Phase Locking (2PL)

- 2PL does not void cascading aborts

T_1	T_2
lock-s(A)	
read(A)	
lock-x(A)	
write(A)	
unlock(A)	
	lock-s(A)
	read(A)
	lock-x(A)
	write(A)
abort	

Two Phase Locking (2PL)

Strict 2PL

- 2PL + hold all exclusive locks till Tx commits or aborts

Rigorous 2PL

- Strict 2PL + hold all shared locks till Tx commits or aborts

Lock Conversions

2PL with lock conversions

- First phase
  - can acquire lock-s
  - can acquire lock-x
  - can convert a lock-s to lock-x (upgrade)
- Second Phase
  - can release lock-s
  - can release lock-x
  - can convert a lock-x to lock-s (downgrade)

Lock Manager

- Separate process (conceptually)
- Maintains a lock table – hash table that records granted locks and pending requests (usually FIFO)

Lock Table (Example)

Data items	Granted set	Mode	Pending requests
A	$\{ T_1, T_2 \}$	lock-s	$T_3(x) \leftarrow T_4(x)$
B	$\{ T_7 \}$	lock-x	$T_5(x) \leftarrow T_6(s)$

- Lock table is a shared data structure (grabbed by the Tx using semaphore/mutex; this is different from 2PL)

Lock Acquisition

When reading A

```
1: if T_i has lock on A then
2:     read(A)
3: else
4:     while !(no other Tx has lock-x on A) do
5:         wait
6:     grant T_i lock-s on A
7:     read(A)
```

When writing A

```
1: if T_i has lock-x on A then
2:     write(A)
3: else
4:     while !(no other Tx has lock on A) do
5:         wait
6:     if T_i has lock-s on A then
7:         upgrade to lock-x
8:     else
9:         grant T_i lock-x on A
10:    write(A)
```

Deadlocks

T_1	T_2	
lock-s(A)		
...		
	lock-x(B)	
	...	
	lock-x(A)	denied; waiting
...		
lock-x(B)		denied; waiting

Deadlock Prevention

- Ensure that the system will never enter into a deadlock

Deadlock prevention protocols

- Pre-declaration: Each Tx locks all its data items before it begins execution
- Order locking requests (possible when single lock upgrade request)
- Graph-based: Impose ordering on data items and only acquire locks based on the partial order specified

Deadlock Avoidance

- Assign priorities p based on age of Tx
- age = `now() - tx.startTime()`

Consider that T_2 wants a lock that T_1 holds

- **Wait-Die**: if $p(T_2) > p(T_1)$, then T_2 waits for T_1, else T_2 aborts
- **Wound-wait**: if $p(T_2) > p(T_1)$, then T_1 aborts, else T_2 waits
- when a transaction is aborted, then use original timestamp instead of `now()` to compute age

Deadlock Detection and Recovery

- Maintain a **waits-for** graph
- Periodically check for cycles in graph

T_1	T_2	T_3	T_4	
lock-s(A)				
lock-s(D)				
	lock-x(B)			
lock-s(B)				denied; wait for T_2
		lock-s(D)		
		lock-s(C)		
	lock-x(C)			denied; wait for T_3
			lock-x(B)	denied; wait for T_2
		lock-x(A)		denied; wait for T_1

Deadlock Detection and Recovery

- To recover,
  - pick a victim to roll back; Tx whose roll back will incur least cost
  - Rollback can either be partial or complete
  - Can lead to starvation, if same Tx is chosen as victim; include #rollback in cost factor to avoid starvation

Locking Extensions

- Multiple granularity locking
- Intention locks
- Hierarchical lock protocol

Timestamp-based Protocol

- Each Tx T_i is assigned a time stamp $TS(T_i)$
- Tx T_j that comes later will have $TS(T_j) > TS(T_i)$

Key Idea

- Time stamp determines the serializability order
- If two Tx execute conflicting instructions in order that violates their timestamp ordering, then one of them has to abort!

T_1	T_2
	write(A)
read(A)	

Not OK

T_1	T_2
write(A)	
	write(A)

OK

T_1	T_2
write(A)	
	read(A)

OK

T_1	T_2
	read(A)
write(A)	

Not OK

Timestamp-based Protocol

Each data item A is associated with two timestamps

1. **W-TS(A)**: largest timestamp of any Tx that wrote A successfully
2. **R-TS(A)**: largest timestamp of any Tx that read A successfully

Protocol

T_i issues a $read(A)$

1: **if** $TS(T_i) <$ W-TS(A) **then**
2: reject read
3: rollback T_i
4: **else**
5: $read(A)$
6: R-TS(A) $\leftarrow \max($R-TS(A), $TS(T_i))$

T_i issues a $write(A)$

1: **if** $TS(T_i) <$ R-TS(A) **then**
2: reject write; rollback T_i
3: **else if** $TS(T_i) <$ W-TS(A) **then**
4: reject write; rollback T_i
5: **else**
6: write(A)
7: W-TS(A) $\leftarrow TS(T_i)$

Timestamp-based Protocol

$TS(T_1) = 1$ and $TS(T_2) = 2$

T_1	T_2
read(B)	
	read(B)
	B:=B-50
read(A)	
	read(A)
print(A+B)	
	A:=A+50
	write(A)
	print(A+B)

This schedule is serializable under the timestamp-based protocol

Timestamp-based Protocol

T_1	T_2	T_3	T_4	T_5
				read(A)
	read(B)			
read(B)				
		write(B)		
		write(C)		
				read(C)
	read(C)			
	abort			
read(A)				
			read(D)	
		write(D)		
		abort		
				write(B)
				write(D)

May not prevent cascading aborts!

Timestamp-based Protocol

T_1	T_2	T_3	T_4	T_5
				read(A)
	read(B)			
read(B)				
		write(B)		
		write(C)		
				read(C)
				commit
	read(C)			
	abort			
read(A)				
			read(D)	
		write(D)		
		abort		

May not be recoverable!

Timestamp-based Protocol

- Guarantees serializability

  Proof sketch: construct precedence graph, all edges are of form $T_i \rightarrow T_j$, where $TS(T_i) < TS(T_j)$; no cycles!

- No deadlocks! (no transaction ever waits)
- Can lead to starvation!

- Schedule may not prevent cascading aborts
- Schedule may not even be recoverable

Timestamp-based Protocol

Extensions to ensure recoverability and cascadelessness

- Perform all writes **atomically** at the end of transaction.
- Use limited form of locking: postpone reads of uncommitted items until Tx that has updated the item commits

Validation-based Protocol

Optimistic Concurrency Control

- Transaction is executed in three phases

1. **Read:** T_i writes only to temporary local variables
2. **Validate:** Determine if local variables can be written without violating serializability
3. **Commit/Rollback:**
   - If validation successful, write operations performed by T_i are copied to database; read-only Tx can skip this phase
   - Else, if there is conflict: resolve by aborting the T_i (other conflict resolution schemes are also possible)

- Guarantees serializability (by timestamp ordering; next slide)
- Validation scheme avoids cascading aborts

Validation-based Protocol

Each Tx T_i is associated with

- **Start-TS(T_i)**: time when T_i started its execution
- **Validation-TS(T_i)**: time when T_i finished its read phase and started its validation phase
- **Finish-TS(T_i)**: time when T_i finished its commit phase

Validation test

- For T_i, for all T_k such that $TS(T_k) < TS(T_i)$ $\quad TS(T_i) = \text{Validation-TS}(T_i)$
  one of the following two conditions must hold

1. $\text{Finish-TS}(T_k) < \text{Start-TS}(T_i)$
2. $\text{W-DataItems}(T_k)$ does not intersect with $\text{R-DataItems}(T_i)$
   and $\text{Start-TS}(T_i) < \text{Finish-TS}(T_k) < \text{Validation-TS}(T_i)$

Validation-based Protocol

T_1	T_2
read(B)	
	read(B)
	B:=B-50
	read(A)
	A:=A+50
read(A)	
validate	
print(A+B)	
	validate
	write(B)
	write(A)

- Schedule is serializable under validation-based protocol

OCC is Widely Used (few examples)

Google App Engine

Kubernetes

Summary so Far

- Transaction = Unit of work (sequence of reads and writes) with all-or-nothing property
- ACID properties
- Concurrency Control
  - Schedules
  - Conflicts
  - Serializability
  - Conflict serializability
  - Recoverability
  - Preventing cascading aborts
- Locking-based protocols (2PL)
- Timestamp-based protocols
- Validation-based protocols (OCC)

Without ACID

Without ACID

ZHONG's Scheme to Defraud

Silk Road was an online "darknet" black market. In operation from approximately 2011 until 2013, Silk Road was used by numerous drug dealers and other unlawful vendors to distribute massive quantities of illegal drugs and other illicit goods and services to many buyers and to launder all funds passing through it. In 2015, following a groundbreaking prosecution by this Office, Silk Road's founder Ross Ulbricht was convicted by a unanimous jury and sentenced to life in prison.

In September 2012, ZHONG executed a scheme to defraud Silk Road of its money and property by (a) creating a string of approximately nine Silk Road accounts (the "Fraud Accounts") in a manner designed to conceal his identity; (b) triggering over 140 transactions in rapid succession in order to trick Silk Road's withdrawal-processing system into releasing approximately 50,000 Bitcoin from its Bitcoin-based payment system into ZHONG's accounts; and (c) transferring this Bitcoin into a variety of separate addresses also under ZHONG's control, all in a manner designed to prevent detection, conceal his identity and ownership, and obfuscate the Bitcoin's source.

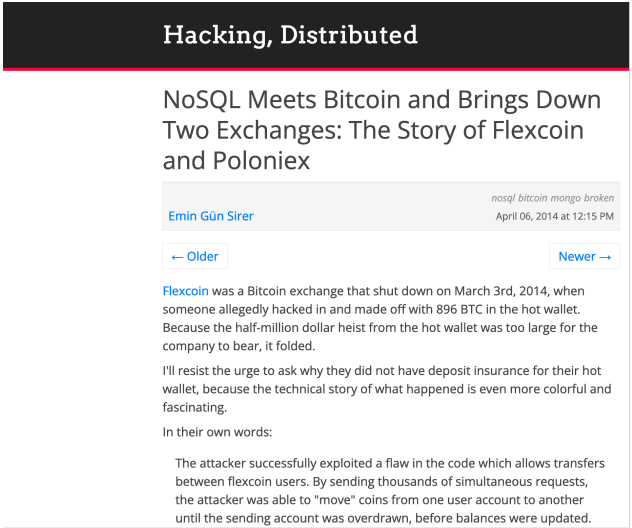
While executing the September 2012 fraud, ZHONG did not list any item or service for sale on Silk Road, nor did he buy any item or service on Silk Road. ZHONG registered the accounts by providing the bare minimum of information required by Silk Road to create the account; the Fraud Accounts were merely a conduit for ZHONG to defraud Silk Road of Bitcoin.

ZHONG funded the Fraud Accounts with an initial deposit of between 200 and 2,000 Bitcoin. After the initial deposit, ZHONG then quickly executed a series of withdrawals. Through his scheme to defraud, ZHONG was able to withdraw many times more Bitcoin out of Silk Road than he had deposited in the first instance. As an example, on September 19, 2012, ZHONG deposited 500 Bitcoin into a Silk Road wallet. Less than five seconds after making the initial deposit, ZHONG executed five withdrawals of 500 Bitcoin in rapid succession — *i.e.*, within the same second — resulting in a net gain of 2,000 Bitcoin. As another example, a different Fraud Account made a single deposit and over 50 Bitcoin withdrawals before the account ceased its activity. ZHONG moved this Bitcoin out of Silk Road and, in a matter of days, consolidated them into two high-value amounts.

Nearly five years after ZHONG's fraud, in August 2017, solely by virtue of ZHONG's possession of the 50,000 Bitcoin that he unlawfully obtained from Silk Road, ZHONG received a matching amount of a related cryptocurrency — 50,000 Bitcoin Cash ("BCH Crime Proceeds") — on top of the 50,000 Bitcoin. In August 2017, in a hard fork coin split, Bitcoin split into two cryptocurrencies, traditional Bitcoin and Bitcoin Cash ("BCH"). When this split occurred, any Bitcoin address that had a Bitcoin balance (as ZHONG's addresses did) now had the exact same balance on both the Bitcoin blockchain and on the Bitcoin Cash blockchain. As of August 2017, ZHONG thus possessed 50,000 BCH in addition to the 50,000 Bitcoin that ZHONG unlawfully obtained from Silk Road. ZHONG thereafter exchanged through an overseas cryptocurrency exchange all of the BCH Crime Proceeds for additional Bitcoin, amounting to approximately 3,500 Bitcoin of additional crime proceeds. Collectively, by the last quarter of 2017, ZHONG thus possessed approximately 53,500 Bitcoin of total crime proceeds (the "Crime Proceeds").

TOP

Without ACID



Hacking, Distributed

NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex

nosql bitcoin mongo broken

Emin Gün Sirer — April 06, 2014 at 12:15 PM

← Older Newer →

Flexcoin was a Bitcoin exchange that shut down on March 3rd, 2014, when someone allegedly hacked in and made off with 896 BTC in the hot wallet. Because the half-million dollar heist from the hot wallet was too large for the company to bear, it folded.

I'll resist the urge to ask why they did not have deposit insurance for their hot wallet, because the technical story of what happened is even more colorful and fascinating.

In their own words:

> The attacker successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, before balances were updated.