Transactions

- Serializability
- Properties
  - recoverability, cascading aborts
- Concurrency control via locks
  - strict, rigorous, intention
- Deadlocks
- Other approaches to serialization
- Recovery

Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of *optimistic* concurrency control
- Key idea:
  - For each object, maintain past "versions" of the data along with timestamps
    - Every update to an object causes a new version to be generated

Snapshot Isolation

- Read queries:
  - Let "t" be the "timestamp" of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of "t"
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
  - Never aborted
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage. However, *the writes are visible to the transaction that made them.*
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

Snapshot Isolation

- *Logically*, T_1 under Snapshot Isolation:
  - takes snapshot of committed data at start
  - only reads/modifies data in local snapshot
  - updates of concurrent transactions not visible to T_1
  - writes of T_1 complete when it commits
  - **First-committer-wins rule:**
    - Commits only if no other concurrent transaction has already written data that T_1 intends to write (*overlapping writesets*)
  - *Or:* **first-writer-wins rule**

initial values zero

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y) → 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible (→ R(Z) → 0)
Own updates are visible (→ R(Y) → 1)
Not first-committer of X (→ W(X:=3))
Serialization error, T2 is rolled back (→ Abort)

Snapshot Isolation

"first committer"

- Advantages:
  - Read queries do not block, never abort
  - Update transactions don't abort *as long as conflicts are rare.*
  - Overall better performance than locking-based protocols
- Major disadvantage:
  - Not serializable!

But: *x = y = 0*

T_1	T_2
w(x)1	
	w(y)1
r(y)0	
	r(x)0
commit?	
	commit?

Snapshot Isolation *implementation via multi-version database*

- High-level:
  - each write to Q creates a new version of Q (old versions retained)
  - reads parameterized by transaction's *timestamp*
    - satisfied by last write before that timestamp
- Timestamp usage:
  - transaction gets *StartTS(T_i)*, *CommitTS(T_i)*,
  - write by T_i saved with *CommitTS(T_i)*
  - read by T_i satisfied by last version w/ time $<$ *StartTS(T_i)*
  - as a result:
    - transaction only see writes committed prior to start
    - i.e. a *snapshot*

Snapshot Isolation *implementation is via multi-version database*

Two validation approaches: *first-committer-wins*, and *first-updater-wins.*

T_j is said to be concurrent with a transaction T_i if timestamps overlap:

$StartTS(T_j) \leq StartTS(T_i) \leq CommitTS(T_j)$, or

$StartTS(T_i) \leq StartTS(T_j) \leq CommitTS(T_i)$

Under *first-committer-wins* (the default), T_i checks at commit time to see if any *concurrent* transaction has written an object that it is trying to write. If so, T_i **aborts**.

Under *first-updater-wins*, T_i checks at each write. Before writing *Q*, T_i :

- Attempts to acquire a write lock on *Q*. If the lock is acquired, T_i **aborts** if a concurrent transaction T_j has already written *Q*.
- If the lock was *not* successful, T_i waits to see if T_j commits or aborts. If T_j commits, T_i **aborts**. If T_j aborts:
  - T_i repeats the check for a concurrent writer having updated *Q*. If found,
    - T_i **aborts.**
  - else
    - T_i **commits**

Snapshot Isolation

- Advantages:
  - Read queries don't block at all, run fast
  - If conflicts rare, update transactions don't abort either
  - Overall better performance than locking protocols
- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation

Transactions

- Serializability
- Properties
  - recoverability, cascading aborts
- Concurrency control via locks
  - strict, rigorous, intention
- Deadlocks
- Other approaches to serialization
- Recovery

Timestamp-Ordering Protocol

- No locks
- Transactions issued timestamps when started
- Timestamps determine the *serializability order*
- If T1 enters before T2, then T1 < T2 in serializability order
- Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger timestamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger timestamp already read, or wrote, that data item, then the write is *rejected* and T1 is aborted
  - Aborted transactions are restarted with a new timestamp
    - Possibility of *starvation*
    - Optimistic

Timestamp-Ordering Protocol

- Example

TS($T1$) < TS($T2$) < TS($T3$) < TS($T4$) < TS($T5$)

T_1	T_2	T_3	T_4	T_5
				write(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X) ***abort***			
read(X) ***abort***				
		write(Z) ***abort***		
				write(Y)
				write(Z)

Timestamp-Ordering Protocol

- The following set of instructions is not conflict-serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- As discussed before, not even *view-serializable:*
  - if T_i reads initial value of Q in S, must also in S'
  - if T_i reads value written from T_j in S, must also in S'
  - if T_i performs final write to Q in S, must also in S'

not both possible at once

Timestamp-Ordering

- Thomas' Write Rule
  - *Ignore obsolete writes*

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Ignored if $T_3 < T_4$

- Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger timestamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger timestamp already read, ~~or wrote~~, that data item, then the write is *rejected* and T1 is aborted
    - *If a transaction with larger timestamp already written that data item, then the write is ignored*

Timestamp-Ordering Protocol

- As discussed here, has a few issues
  - Starvation
  - Non-recoverable
  - Cascading rollbacks possible
- Most can be solved fairly easily
  - Read up
- We can always add more restrictions to ensure these things
  - The goal is to find the minimal set of restrictions to as to not hinder concurrency

Validation Protocol

- Each transaction T_i has 3 timestamps
  - Start(T_i) : when T_i starts execution
  - Validation(T_i): when T_i enters its validation phase
  - Finish(T_i) : when T_i finishes its write phase
- Serializability order = validation order
  - TS(T_i) = Validation(T_i)
  - increases concurrency.
- Higher degree of concurrency if conflicts low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Protocol

If for all T_k with TS(T_k) < TS(T_i) then validation of $\boldsymbol{T_i}$ succeeds if:

- finish(T_k) < start(T_i)

or:

- the set of data items written by T_k does not intersect with the set of data items read by T_i and
- T_k completes its write phase before T_i starts validation:

  start(T_i) < finish(T_k) < validation(T_i)

Validation Protocol

- Serialization order?
  - $T_{25} < T_{26}$
- T_{25} validates?
  - because first
- T_{26} validates?
  - T_{25} did not write

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
<validate>	
display($A + B$)	
	<validate>
	write(B)
	write(A)

- finish(T_k) < start(T_i)

or:

- data items written by T_k do not intersect with data items read by T_i and
- start(T_i) < finish(T_k) < validation(T_i)

Validation Protocol

x = 0

T_1	T_2	T_3
<start>		
write(X) = 1		
	<start>	
	read(X) = 0	
<validate>		
<commit>		
	<validate>	<start>
	<abort>	read(X) = 1
		<validate>
		<commit>

- finish(T_k) < start(T_i)

or:

- data items written by T_k do not intersect with data items read by T_i and
- start(T_i) < finish(T_k) < validation(T_i)

Weak Levels of Isolation in SQL

- SQL can be parameterized by isolation level:
  - **Read uncommitted**: allows *uncommitted writes* to be read
  - **Read committed**: only read committed data, repeated reads of same data might return different values as other transactions commit
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value
    - so read locks should be retained or caching used
    - transaction-local writes can change subsequent reads
    - Phantom problem not necessarily prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Serializable**: default, strongest (except for *linearizable*)
- In many database systems, *read committed is the default*
  - has to be explicitly changed to serializable when required
    - `set isolation level serializable`
- Oracle calls snapshot isolation “serializable”

Weak Isolation Levels: Read Uncommitted

x = y = 0

T_1	T_2	T_3
start		
write(X) = 1		
write(Y) = 1		
		start
		read(X) = 1
		write(X) = 3
commit		
		read(X) = 3
	start	
	write(X) = 2	
	commit	
		read(X) = 2
		read(Y) = 1
		commit

- Not serializable
- Doesn’t guarantee recoverable scheds
- Not free from cascading aborts

Weak Isolation Levels: Read Committed

x = y = 0	T_1	T_2	T_3
	start		
	write(X) = 1		
	write(Y) = 1		start
			read(X) = 0
			write(X) = 3
	commit		
			read(X) = 3
		start	
		write(X) = 2	
		commit	
			read(X) = 2
			read(Y) = 1
			commit

- Not serializable
- Guarantees recoverable scheds
- Free from cascading
- Stronger isolation than read uncommitted

Weak Isolation Levels: Repeatable Reads

x = y = 0	T_1	T_2	T_3
	start		
	write(X) = 1		
			start
			read(X) = 0
	commit		
			read(X) = 0
		start	
		write(X) = 2	
		commit	
			read(X) = 0
			commit

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Still not serializable, but even stronger isolation

Weak Isolation Levels: Snapshot Iso

x = y = 0	T_1	T_2	T_3
	start		
	write(X) = 1		
	read(Y) = 0		
		start	
		write(Y) = 2	
		read(X) = 0	
			start
			write(Y) = 3
			read(X) = 0
		commit	
			abort
	commit		

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Faster

strict Serializability

x = 0	T_1	T_2	T_3
	start		
	write(X) = 1		
	write(Y) = 1		start
			wait
			wait
	commit		*wait*
			read(X) = 1
			write(X) = 3
			read(X) = 3
			read(X) = 3
			read(Y) = 1
			commit
		start	
		write(X) = 2	
		commit	

- Locking makes it very different