Transactions

Alan Fekete (U of Sydney)
fekete@it.usyd.edu.au

Overview

- Transactions
  - Concept
  - ACID properties
  - Examples and counter-examples
- Implementation techniques
- Weak isolation issues

Definition

- A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition
  - In the real world, this happened (completely) or it didn't happen at all (Atomicity)
- Commerce examples
  - Transfer money between accounts
  - Purchase a group of products
- Student record system
  - Register for a class (either waitlist or allocated)

Coding a transaction

- Typically a computer-based system doing OLTP has a collection of *application programs*
- Each program is written in a high-level language, which calls DBMS to perform individual SQL statements
  - Either through embedded SQL converted by preprocessor
  - Or through Call Level Interface where application constructs appropriate string and passes it to DBMS

Why write programs?

- Why not just write a SQL statement to express "what you want"?
- An individual SQL statement can't do enough
  - It can't update multiple tables
  - It can't perform complicated logic (conditionals, looping, etc)

COMMIT

- As app program is executing, it is "in a transaction"
- Program can execute COMMIT
  - SQL command to finish the transaction successfully
  - The next SQL statement will automatically start a new transaction

Warning

- The idea of a transaction is hard to see when interacting directly with DBMS, instead of from an app program
- Using an interactive query interface to DBMS, by default each SQL statement is treated as a separate transaction (with implicit COMMIT at end) unless you explicitly say “START TRANSACTION”

A Limitation

- Some systems rule out having both DML and DDL statements in a single transaction
- i.e., you can change the schema, or change the data, but not both

ROLLBACK

- If the app gets to a place where it can’t complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when app program finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

Atomicity

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made
- That is, transaction’s activities are all or nothing

Integrity

- A real world state is reflected by collections of values in the tables of the DBMS
- But not every collection of values in a table makes sense in the real world
- The state of the tables is restricted by integrity constraints
- e.g. account number is unique
- e.g. stock amount can’t be negative

Integrity (ctd)

- Many constraints are explicitly declared in the schema
  – So the DBMS will enforce them
  – Especially: primary key (some column's values are non null, and different in every row)
  – And referential integrity: value of foreign key column is actually found in another "referenced" table
- Some constraints are not declared
  – They are business rules that are supposed to hold

Consistency

- Each transaction can be written on the assumption that all integrity constraints hold in the data, before the transaction runs
- It must make sure that its changes leave the integrity constraints still holding
  – However, there are allowed to be intermediate states where the constraints do not hold
- A transaction that does this, is called consistent
- This is an obligation on the programmer
  – Usually the organization has a testing/checking and sign-off mechanism before an application program is allowed to get installed in the production system

System obligations

- Provided the app programs have been written properly,
- Then the DBMS is supposed to make sure that the state of the data in the DBMS reflects the real world accurately, as affected by all the committed transactions

Local to global reasoning

- Organization checks each app program as a separate task
  – Each app program running on its own moves from state where integrity constraints are valid to another state where they are valid
- System makes sure there are no nasty interactions
- So the final state of the data will satisfy all the integrity constraints

Example - Tables

- System for managing inventory
- InStore(prodID, storeID, qty)
- Product(prodID, desc, mnfr, …, WarehouseQty)
- Order(orderNo, prodID, qty, rcvd, ….)
  – Rows never deleted!
  – Until goods received, rcvd is null
- Also Store, Staff, etc etc

Example - Constraints

- Primary keys
  – InStore: (prodID, storeID)
  – Product: prodID
  – Order: orderId
  – etc
- Foreign keys
  – Instore.prodID references Product.prodID
  – etc

Example - Constraints

- Data values
  - Instore.qty >= 0
  - Order.rcvd <= current_date or Order.rcvd is null
- Business rules
  - for each p, (Sum of qty for product p among all stores and warehouse) >= 50
  - for each p, (Sum of qty for product p among all stores and warehouse) >= 70 or there is an outstanding order of product p

Example - transactions

- MakeSale(store, product, qty)
- AcceptReturn(store, product, qty)
- RcvOrder(order)
- Restock(store, product, qty)
  - // move from warehouse to store
- ClearOut(store, product)
  - // move all held from store to warehouse
- Transfer(from, to, product, qty)
  - // move goods between stores

Example - ClearOut

- Validate Input (appropriate product, store)
- SELECT qty INTO :tmp
  FROM InStore
  WHERE StoreID = :store AND prodID = :product
- UPDATE Product
  SET WarehouseQty = WarehouseQty + :tmp
  WHERE prodID = :product
- UPDATE InStore
  SET Qty = 0
  WHERE prodID = :product
- COMMIT

This is one way to write the application; other algorithms are also possible

Example - Restock

- Input validation
  - Valid product, store, qty
  - Amount of product in warehouse >= qty
- UPDATE Product
  SET WarehouseQty = WarehouseQty - :qty
  WHERE prodID = :product
- If no record yet for product in store
  INSERT INTO InStore (:product, :store, :qty)
- Else, UPDATE InStore
  SET qty = qty + :qty
  WHERE prodID = :product and storeID = :store
- COMMIT

Example - Consistency

- How to write the app to keep integrity holding?
- MakeSale logic:
  - Reduce Instore.qty
  - Calculate sum over all stores and warehouse
  - If sum < 50, then ROLLBACK // Sale fails
  - If sum < 70, check for order where date is null
    - If none found, insert new order for say 25
  - COMMIT

This terminates execution of the program (like return)

Example - Consistency

- We don't need any fancy logic for checking the business rules in Restock, ClearOut, Transfer
  - Because sum of qty not changed; presence of order not changed
    - *provided integrity holds before txn, it will still hold afterwards*
- We don't need fancy logic to check business rules in AcceptReturn
  - *why?*
- Is checking logic needed for RcvOrder?

Threats to data integrity

- Need for application rollback
- System crash
- Concurrent activity

- The system has mechanisms to handle these

Application rollback

- A transaction may have made changes to the data before discovering that these aren't appropriate
  - the data is in state where integrity constraints are false
  - Application executes ROLLBACK
- System must somehow return to earlier state
  - Where integrity constraints hold
- So aborted transaction has no effect at all

Example

- While running MakeSale, app changes InStore to reduce qty, then checks new sum
- If the new sum is below 50, txn aborts
- System must change InStore to restore previous value of qty
  - Somewhere, system must remember what the previous value was!

System crash

- At time of crash, an application program may be part-way through (and the data may not meet integrity constraints)
- Also, buffering can cause problems
  - Note that system crash loses all buffered data, restart has only disk state
  - Effects of a committed txn may be only in buffer, not yet recorded in disk state
  - Lack of coordination between flushes of different buffered pages, so even if current state satisfies constraints, the disk state may not

Example

- Suppose crash occurs after
  - MakeSale has reduced InStore.qty
  - found that new sum is 65
  - found there is no unfilled order
  - // but before it has inserted new order
- At time of crash, integrity constraint did not hold
- Restart process must clean this up (effectively aborting the txn that was in progress when the crash happened)

Concurrency

- When operations of concurrent threads are interleaved, the effect on shared state can be unexpected
- Well known issue in operating systems, thread programming
  - see OS textbooks on critical section
  - Java use of synchronized keyword

Famous anomalies

- Dirty data
  - One task T reads data written by T' while T' is running, then T' aborts (so its data was not appropriate)
- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Inconsistent read
  - One task T sees some but not all changes made by T'
  - The values observed may not satisfy integrity constraints
  - This was not considered by the programmer, so code moves into absurd path

Example – Dirty data

- AcceptReturn(p1,s1,50) MakeSale(p1,s2,65)
- Update row 1: 25 -> 75
- update row 2: 70->5
- find sum: 90
- // no need to insert
- // row in Order
- Abort
- // rollback row 1 to 25
- COMMIT

p1	s1	25
p1	s2	70
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Initial state of InStore, Product

p1	s1	25
p1	s2	5
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Integrity constraint is false: Sum for p1 is only 40!

Final state of InStore, Product

Example – Lost update

- ClearOut(p1,s1) AcceptReturn(p1,s1,60)
- Query InStore; qty is 25
- Add 25 to WarehouseQty: 40->65
- Update row 1: 25->85
- Update row 1, setting it to 0
- COMMIT
- COMMIT

p1	s1	25
p1	s2	50
p2	s1	45
etc	etc	etc

p1	etc	40
p2	etc	55
etc	etc	etc

Initial state of InStore, Product

p1	s1	0
p1	s2	50
p2	s1	45
etc	etc	etc

p1	etc	65
p2	etc	55
etc	etc	etc

60 returned p1's have vanished from system; total is still 135

Final state of InStore, Product

Example – Inconsistent read

- ClearOut(p1,s1) MakeSale(p1,s2,60)
- Query InStore: qty is 30
- Add 30 to WarehouseQty: 10->40
- update row 2: 65->5
- find sum: 75
- // no need to insert
- // row in Order
- Update row 1, setting it to 0
- COMMIT
- COMMIT

p1	s1	30
p1	s2	65
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Initial state of InStore, Product

p1	s1	0
p1	s2	5
p2	s1	60
etc	etc	etc

p1	etc	40
p2	etc	44
etc	etc	etc

Integrity constraint is false: Sum for p1 is only 45!

Final state of InStore, Product

Serializability

- To make isolation precise, we say that an execution is serializable when
- There exists some serial (ie batch, no overlap at all) execution of the same transactions which has the same final state
  - Hopefully, the real execution runs faster than the serial one!
- NB: different serial txn orders may behave differently; we ask that *some* serial order produces the given state
  - Other serial orders may give different final states

Example – Serializable execution

- ClearOut(p1,s1) MakeSale(p1,s2,20)
- Query InStore: qty is 30
- update row 2: 45->25
- find sum: 65
- no order for p1 yet
- Add 30 to WarehouseQty: 10->40
- Update row 1, setting it to 0
- COMMIT
- Insert order for p1
- COMMIT

p1	s1	30
p1	s2	45
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Order: empty

Initial state of InStore, Product, Order

p1	s1	0
p1	s2	25
p2	s1	60
etc	etc	etc

p1	etc	40
p2	etc	44
etc	etc	etc

p1	25	Null	etc

Execution is like serial MakeSale; ClearOut

Final state of InStore, Product, Order

Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Treat the set of all serializable executions as an object of interest (called SR)
  - Thm: SR is in NP, i.e. the task of testing whether an execution is serializable seems unreasonably slow
- Does it matter?
  - The goal of practical importance is to design a system that produces some subset of the collection of serializable executions
  - It's not clear that we care about testing arbitrary executions that don't arise in our system

Conflict serializability

- There is a nice sufficient condition (ie a conservative approximation) called conflict serializable, which can be efficiently tested
  - Draw a precedes graph whose nodes are the transactions
  - Edge from Ti to Tj when Ti accesses x, then later Tj accesses x, and the accesses conflict (not both reads)
  - The execution is conflict serializable iff the graph is acyclic
- Thm: if an execution is conflict serializable then it is serializable
  - Pf: the serial order with same final state is any topological sort of the precedes graph
- Most people and books use the approximation, usually without mentioning it!

ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Big Picture

- If programmer writes applications so each txn is consistent
- And DBMS provides atomic, isolated, durable execution
  - i.e. actual execution has same effect as some serial execution of those txns that committed (but not those that aborted)
- Then the final state will satisfy all the integrity constraints

NB true even though system does not know all integrity constraints!

Overview

- Transactions
- Implementation Techniques
  - Ideas, not details!
  - Implications for application programmers
  - Implications for DBAs
- Weak isolation issues

Main implementation techniques

- Logging
  - Interaction with buffer management
  - Use in restart procedure
- Locking
- Distributed Commit

Logging

- The log is an append-only collection of entries, showing all the changes to data that happened, in order as they happened
- e.g. when T1 changes qty in row 3 from 15 to 75, this fact is recorded as a log entry
- Log also shows when txns start/commit/abort

A log entry

- LSN: identifier for entry, increasing values
- Txn id
- Data item involved
- Old value
- New value
  - Sometimes there are separate logs for old values and new values

Extra features

- Log also records changes made by system itself
  - e.g. when old value is restored during rollback
- Log entries are linked for easier access to past entries
  - Link to previous log entry
  - Link to previous entry for the same txn

Buffer management

- Each page has place for LSN of most recent change to that page
- When a page is fetched into buffer, DBMS remembers latest LSN at that time
- Log itself is produced in buffer, and flushed to disk (appending to previously flushed parts) from time to time
- Important rules govern when buffer flushes can occur, relative to LSNs involved
  - Sometimes a flush is forced (eg log flush forced when txn commits)

Using the log

- To rollback txn T
  - Follow chain of T's log entries, *backwards*
  - For each entry, restore data to old value, and produce new log record showing the restoration
  - Produce log record for "abort T"

Restart

- After a crash, follow the log *forward*, replaying the changes
  - i.e. re-install new value recorded in log
- Then rollback all txns that were active at the end of the log
- Now normal processing can resume

Optimizations

- Use LSNs recorded in each page of data, to avoid repeating changes already reflected in page
- Checkpoints: flush pages that have been in buffer too long
  - Record in log that this has been done
  - During restart, only repeat history since last (or second-last) checkpoint

Don't be too confident

- Crashes can occur during rollback or restart!
  - Algorithms must be idempotent
- Must be sure that log is stored separately from data (on different disk array; often replicated off-site!)
  - In case disk crash corrupts data, log allows fixing this
  - Also, since log is append-only, don't want have random access to data moving disk heads away

Complexities

- Multiple txns affecting the same page of disk
  - From "fine-grained locking" (see later)
- Operations that affect multiple pages
  - Eg B-tree reorganization
- Multithreading in log writing
  - Use standard OS latching to prevent different tasks corrupting the log's structure

ARIES

- Until 1992, textbooks and research papers described only simple logging techniques that did not deal with complexities
- Then C. Mohan (IBM) published a series of papers describing ARIES algorithms
  - Papers are very hard to read, give inconsistent level of details, but at last the ideas of modern, high-performance, real systems are available!

Implications

- For application programmer
  - Choose txn boundaries to include everything that must be atomic
  - Use ROLLBACK to get out from a mess
- For DBA
  - Tune for performance: adjust checkpoint frequency, amount of buffer for log, etc
  - Look after the log!

Main implementation techniques

- Logging
- Locking
  - Lock manager
  - Lock modes
  - Granularity
  - User control
- Distributed Commit

Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which data, in which modes
- It rejects a request to get a new lock if a conflicting lock is already held by a different txn
- NB: a lock does not actually prevent access to the data, it only prevents getting a conflicting lock
  - So data protection only comes if the right lock is requested before every access to the data

Lock modes

- Locks can be for writing (W), reading (R) or other modes
- Standard conflict rules: two W locks on the same data item conflict, so do one W and one R lock on the same data
  - However, two R locks do not conflict
- Thus W=exclusive, R=shared

Automatic lock management

- DBMS requests the appropriate lock whenever the app program submits a request to read or write a data item
- If lock is available, the access is performed
- If lock is not available, the whole txn is blocked until the lock is obtained
  - After a conflicting lock has been released by the other txn that held it

Strict two-phase locking

- Locks that a txn obtains are kept until the txn completes
  - Once the txn commits or aborts, then all its locks are released (as part of the commit or rollback processing)
- Two phases:
  - Locks are being obtained (while txn runs)
  - Locks are released (when txn finished)

Serializability

- If each transaction does strict two-phase locking (requesting all appropriate locks), then executions are serializable
- However, performance does suffer, as txns can be blocked for considerable periods
  - Deadlocks can arise, requiring system-initiated aborts

Proof sketch

- Suppose all txns do strict 2PL
- If Ti has an edge to Tj in the precedes graph
  - That is, Ti accesses x before Tj has conflicting access to x
  - Ti has lock at time of its access, Tj has lock at time of its access
  - Since locks conflict, Ti must release its lock before Tj's access to x
  - Ti completes before Tj accesses x
  - Ti completes before Tj completes
- So the precedes graph is subset of the (acyclic) total order of txn commit
- Conclusion: *the execution has same final state as the serial execution where txns are arranged in commit order*

Example – No Dirty data

- AcceptReturn(p1,s1,50) MakeSale(p1,s2,65)
- Update row 1: 25 -> 75
- //t1 W-locks InStore. row 1
- update row 2: 70->5
- //t2 W-locks Instore.row2
- try find sum:// blocked
- // as R-lock on Instore.row1
- // can't be obtained
- User-initiated Abort
- // rollback row 1 to 35; release lock
- // now get locks
- find sum: 40
- ROLLBACK
- // row 2 restored to 70
-

p1	s1	25
p1	s2	70
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Initial state of InStore, Product

p1	s1	25
p1	s2	70
p2	s1	60
etc	etc	etc

p1	etc	10
p2	etc	44
etc	etc	etc

Final state of InStore, Product

Integrity constraint is valid

Example – No Lost update

- ClearOut(p1,s1) AcceptReturn(p1,s1,60)
- Query InStore; qty is 25
- //t1 R-lock InStore.row1
- Add 25 to WarehouseQty: 40->65
- // t1 W-lock Product.row 1
- try Update row 1
- // blocked
- // as W-lock on InStore.row1
- // can't be obtained
- Update row 1, setting it to 0
- //t1 upgrades to W-lock on InStore.row1
- COMMIT // release t1's locks
- // now get W-lock
- Update row 1: 0->60
- COMMIT

p1	s1	25
p1	s2	50
p2	s1	45
etc	etc	etc

p1	etc	40
p2	etc	55
etc	etc	etc

Initial state of InStore, Product

p1	s1	60
p1	s2	50
p2	s1	45
etc	etc	etc

p1	etc	65
p2	etc	55
etc	etc	etc

Final state of InStore, Product

Outcome is same as serial ClearOut; AcceptReturn

Granularity

- What is a data item (on which a lock is obtained)?
  - Most times, in most modern systems: item is one tuple in a table
  - Sometimes: item is a page (with several tuples)
  - Sometimes: item is a whole table
- In order to manage conflicts properly, system gets "intention" mode locks on larger granules before getting actual R/W locks on smaller granules

Granularity trade-offs

- Larger granularity: fewer locks held, so less overhead; but less concurrency possible
  - "false conflicts" when txns deal with different parts of the same item
- Smaller "fine" granularity: more locks held, so more overhead; but more concurrency is possible
- System usually gets fine grain locks until there are too many of them; then it replaces them with larger granularity locks

Explicit lock management

- With most DBMS, the application program can include statements to set or release locks on a table
  - Details vary
- e.g. LOCK TABLE InStore IN EXCLUSIVE MODE

Implications

- For application programmer
  - If txn reads many rows in one table, consider locking the whole table first
  - Consider weaker isolation (see later)
- For DBA
  - Tune for performance: adjust max number of locks, granularity factors
  - Possibly redesign schema to prevent unnecessary conflicts
  - Possibly adjust query plans if locking causes problems

Implementation mechanisms

- Logging
- Locking
- Distributed Commit

Transactions across multiple DBMS

- Within one transaction, there can be statements executed on more than one DBMS
- To be atomic, we still need all-or-nothing
- That means: every involved system must produce the same outcome
  - All commit the txn
  - Or all abort it

Why it's hard

- Imagine sending to each DBMS to say "commit this txn T now"
- Even though this message is on its way, any DBMS might abort T spontaneously
  - e.g. due to a system crash

NB unrelated to "two-phase locking"

Two-phase commit

- The solution is for each DBMS to first move to a special situation, where the txn is "prepared"
- A crash won't abort a prepared txn, it will leave it in prepared state
  - So all changes made by prepared txn must be recovered during restart (including any locks held before the crash!)

Basic idea

- Two round-trips of messages
  - Request to prepare/ prepared or aborted
  - Either Commit/committed or Abort/aborted

Only if all DBMSs are already prepared!

Read-only optimisation

- If a txn has involved a DBMS only for reading (but no modifications at that DBMS), then it can drop out after first round, without preparing
  - The outcome doesn't matter to it!
  - Special phase 1 reply: ReadOnly

Fault-tolerant protocol

- The interchange of messages between the "coordinator" (part of the TPMonitor software) and each DBMS is tricky
  - Each participant must record things in log at specific times
  - But the protocol copes with lost messages, inopportune crashes etc

Implications

- For application programmer
  - Avoid putting modifications to multiple databases in a single txn
    - Performance suffers a lot
    - W-Locks are held during the message exchanges, which take much longer than usual txn durations
- For DBA
  - Monitor performance carefully
  - Make sure you have DBMS that support protocol

Overview

- Transactions
- Implementation techniques
- Weak isolation issues
  - Explicit use of low levels
  - Use of replicas
  - Snapshot isolation

Problems with serializability

- The performance reduction from isolation is high
  - Transactions are often blocked because they want to read data that another txn has changed
- For many applications, the accuracy of the data they read is not crucial
  - e.g. overbooking a plane is ok in practice
  - e.g. your banking decisions would not be very different if you saw yesterday's balance instead of the most up-to-date

A and D matter!

- Even when isolation isn't needed, no one is willing to give up atomicity and durability
  - These deal with modifications a txn makes
  - Writing is less frequent than reading, so log entries and write locks are considered worth the effort

Explicit isolation levels

- A transaction can be declared to have isolation properties that are less stringent than serializability
  - However SQL standard says that default should be serializable (also called "level 3 isolation")
  - In practice, most systems have weaker default level, and most txns run at weaker levels!

Browse

- SET TRANACTION ISOLATION LEVEL READ UNCOMMITTED
  - Do not set read locks at all
    - Of course, still set write locks before updating data
    - If fact, system forces the txn to be read-only unless you say otherwise
  - Allows txn to read dirty data (from a txn that will later abort)

Cursor stability

- SET TRANACTION ISOLATION LEVEL READ COMMMITTED Most common in practice!
  - Set read locks but release them after the read has happened
    - e.g. when cursor moves onto another element during scan of the results of a multirow query
  - i.e. do not hold R-locks till txn commits/aborts
  - Data is not dirty, but it can be inconsistent (between reads of different items, or even between one read and a later one of the same item)
    - Especially, weird things happen between different rows returned by a cursor

Repeatable read

- SET TRANACTION ISOLATION LEVEL REPEATABLE READ
  - Set read locks on data items, and hold them till txn finished, but release locks on indices as soon as index has been examined
  - Allows “phantoms”, rows that are not seen in a query that ought to have been (or vice versa)
  - Problems if one txn is changing the set of rows that meet a condition, while another txn is retrieving that set

Stale replicas

- In many distributed processing situations, copies of data are kept at several sites
  - e.g. to allow cheap/fast local reading
- If updates try to alter all replicas, they become very slow and expensive (they need two-phase commit, and they’ll abort if a remote site is unavailable!)
- So allow replicas to be out-of-date
- Lazy propagation of updates
  - Easily managed by shipping the log across from time to time

Reading stale replicas

- If a txn reads a local replica which is a bit stale, then the value read can be out-of-date, and potentially inconsistent with other data seen by the txn
- Impact is essentially the same as READ COMMITTED

Snapshot Isolation

- Most DBMS vendors use variants of the standard algorithms
- However, one very major vendor uses a different approach: Oracle
  - Before version 7.3 it did not support ISOLATION LEVEL SERIALIZABLE at all
  - Now it allows the SQL command, but uses a different algorithm called Snapshot Isolation

Snapshot Isolation

- Read of an item does not give current value
- Instead, use the recovery log to find value that had been most recently committed at the time the txn started
  - Exception: if the txn has modified the item, use the value it wrote itself
- The transaction sees a "snapshot" of the database, at an earlier time
  - Intuition: this should be consistent, if the database was consistent before

Checks for conflict

- If two overlapping txns try to modify the same item, one will be aborted
- Implemented with write locks on modified rows
  - NB one txn out of the conflicting pair is aborted, rather than delayed as in conventional approach

Benefits of SI

- No cost for extra time-travel versions
  - They are in log anyway!
- Reading is *never* blocked
- Prevents the usual anomalies
  - No dirty read
  - No lost update
  - No inconsistent read

Problems with SI

- SI does not always give serializable executions
  - (despite Oracle using it for "ISOLATION LEVEL SERIALIZABLE)
- Integrity Constraints can be violated
  - Even if every application is written to be consistent!

NB: sum uses old value of row1 and Product, and self-changed value of row2

Example – Skew Write

- MakeSale(p1,s1,26) MakeSale(p1,s2,25)
- Update row 1: 30->4
- update row 2: 35->10
- find sum: 72
- // No need to Insert row in Order
- Find sum: 71
- // No need to insert row in Order
- COMMIT
- COMMIT

p1	s1	30
p1	s2	35
p2	s1	60
etc	etc	etc

p1	etc	32
p2	etc	44
etc	etc	etc

Order: empty

Initial state of InStore, Product, Order

p1	s1	4
p1	s2	10
p2	s1	60
etc	etc	etc

p1	etc	32
p2	etc	44
etc	etc	etc

Order: empty

Final state of InStore, Product, Order

Integrity constraint is false: Sum is 46

Skew Writes

- SI breaks serializability when txns modify different items, each based on a previous state of the item the other modified
- This is fairly rare in practice
- Eg the TPC-C benchmark runs correctly under SI
  - when txns conflict due to modifying different data, there is also a shared item they both modify too (like a total quantity) so SI will abort one of them

Implications

- For the application programmer
  - Think carefully about your programs behavior if reads are inaccurate
  - If possible without compromising correctness, run at lower isolation level to improve performance
- For the DBA
  - Watch like a hawk for corruption of the data, and have strong processes to correct it!

Further Reading

- Transaction concept: Standard database texts, e.g. Garcia-Molina et al Chapter 8.6
- Main implementation techniques: e.g. Garcia-Molina et al Chapters 17-19
- Big picture: “Principles of Transaction Processing” by P. Bernstein and E. Newcomer
- Theory: “Transactional Information Systems” by G. Weikum and G. Vossen
- The gory details: “Transaction Processing” by J. Gray and A. Reuter

Recent Transaction Research

- Properties of weak isolation
  - Declarative representation
  - Restricted cases where you still get integrity running with lower isolation level
    - Conditions on the applications
    - Conditions on the constraints
- Extended transaction models
  - Suitable for web services workflows
  - Across trust domains, so can’t give up autonomy