Transactions

CSEP 544
Monday April 5, 2004
Alan Fekete (U of Sydney)

Overview

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

Further Reading

- Transaction concept: Garcia-Molina et al. Chapter 8.6
- Implementation techniques: Garcia-Molina et al. Chapters 17-19
- Big picture: "Principles of Transaction Processing" by P. Bernstein and E. Newcomer
- The gory details: "Transaction Processing" by J. Gray and A. Reuter

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 called DBMS to perform individual SQL statements
  - Either through embedded SQL converted by preprocessor
  - Or through Call-Level Interface where application constructs a 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 complex 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"

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)
- App program finds a problem
  - Eq. qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - Eg due to time outs

Atomicity

- Two possible outcomes for a transaction
  - If commits: all the changes are made
  - If 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.
  - Eg. account number is unique.
  - Eg. 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 also ensure that the 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 program writer.
  - Usually, the organization has a testing/checking and sign-off schedule, before an application program is allowed to get installed in the production system.

System obligations

- Provided the application 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 in an isolated 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, ...)
  - Row is 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 - Data values

- `Instore.qty` >= 0
- `Order.rcvd` <= current_date or `Order.rcvd` is null

Example - 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 - CleanOut

- 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`

Example - Restock

- Input validation
  - Valid product, store, qty
  - Arm count 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
    - `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 orders where date is null
      - If none found, insert new order for say 25

Example - Consistency

- We don’t need any fancy logic for the business rules in AcceptReturn, Restock, ClearOut, Transfer
  - Why?
- What is 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, bm aborts
- System must change InStore to restore previous value of qty
  - Somewhere, system must remember what the previous value was!

System crash

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

Example

- Suppose crash occurs after
  - MakeSale has reduced InStore qty
  - found that new qty is 65
- found there is no unfilled order
  - before it has inserted new order
- At the end of crash, integrity constraint did not hold
- Restart process must clean this up (effectively aborting the transaction 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 is 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 assumes a broken state changes done by T’.
  - The values observed are not exactly integrity constraints.
  - This is not considered by the program and 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 35
- COMMIT (etc etc etc)

Initial state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
<tr>
<td>s1</td>
<td>60</td>
</tr>
<tr>
<td>s2</td>
<td>70</td>
</tr>
</tbody>
</table>

Final state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
</tbody>
</table>

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

Example - Lost update

- ClearOut(p1,s1)
  - AcceptReturn(p1,s1,60)
  - Query InStore; qty is 25
  - Add 25 to WarehouseQty: 10 -> 40
  - Update row 1: 25 -> 85
  - Update row 1, setting it to 0
- COMMIT
- COMMIT

Initial state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
<tr>
<td>s1</td>
<td>60</td>
</tr>
<tr>
<td>s2</td>
<td>70</td>
</tr>
</tbody>
</table>

Final state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
</tbody>
</table>

60 returned pl’s have vanished from system; total is still 135

Example - Inconsistent read

- ClearOut(p1,s1)
  - MakeSale(p1,s2,60)
  - Query InStore; qty is 25
  - Add 25 to WarehouseQty: 10 -> 40
  - Update row 1: 25 -> 85
  - Update row 1, setting it to 0
- COMMIT
- COMMIT

Initial state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
<tr>
<td>s1</td>
<td>60</td>
</tr>
<tr>
<td>s2</td>
<td>70</td>
</tr>
</tbody>
</table>

Final state of InStore, Product

<table>
<thead>
<tr>
<th>id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>25</td>
</tr>
<tr>
<td>p2</td>
<td>60</td>
</tr>
</tbody>
</table>

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

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 order means may behave differently; we ask that some serial order produces the given state.
  - Other serial orders may give different final states.
Example - Serializable execution

Serializability Theory

ACID

Overview

Main implementation techniques

Big Picture

Overview

Main implementation techniques

ACID

- Atomic
  - State shows either all the effects of txns, 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 if txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Overview

- Transactions
- Implementation Techniques
  - Ideas, not details!
  - Implementations for application programs
  - Implementations 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 field qty in row 3 from 15 to 75, this fact is recorded as a log entry.
- Log also shows when transactions 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 transaction.

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 (e.g., log flush forced when transaction commits).

Using the log

- To rollback transaction 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 transactions 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!
  - Algorithm must be idempotent
- Must be sure that log is stored separately from data (in different disk array; often replicated off-site!)
  - In case disk crash corrupts data, log allows fixing this
  - As log is append-only, don’t want random access to data in case disk heads away

Complexities

- Changes to index structures
  - Avoid logging every time index is rearranged
- 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, and omit crucial details, but at least the ideas of real systems are now 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 remember which transactions 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 transaction.
- NB: a lock does not actually prevent access to the data, it only prevents getting a conflicting lock.
  - So data protection only ever as 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 application submits a request to read or write a data item.
- If the lock is available, the access is performed.
- If the lock is not available, the whole transaction is blocked until the lock is obtained.
  - After a conflicting lock has been released by the other transaction that held it.

Strict two-phase locking

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

Serializability

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

Example - No Dirty data

```
Initial state of InStore, Product:
60s1p2
70s2p1
25s1p1

Initial state of InStore, Product:
44s1p2
10s2p1
```

Integrity constraint is valid
Example – No Lost update

- ClearOut(p1) AcceptReturn(p1,s1)
- Query InStore, qty is 25
- Add 25 to W writtenQty, qty is 45
- Update row 1, qty is 45
  - Blocked
  - W lock on InStore row 1
  - Can’t be obtained
- Update row 1, qty is 0
  - W upgrade to W lock on InStore row 1
  - COMMIT // release W’s lock
  - Update row 1, qty is 60
- COMMIT

Initial state of InStore, Product:

<table>
<thead>
<tr>
<th>p1</th>
<th>s1</th>
<th>p2</th>
<th>s2</th>
<th>qty</th>
<th>addr</th>
<th>dte</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td></td>
<td>25</td>
<td></td>
<td>65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>25</td>
<td></td>
<td>65</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Final state of InStore, Product:

<table>
<thead>
<tr>
<th>p1</th>
<th>s1</th>
<th>p2</th>
<th>s2</th>
<th>qty</th>
<th>addr</th>
<th>dte</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td></td>
<td>60</td>
<td></td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>60</td>
<td></td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Outcome is same as serial ClearOut, AcceptReturn etc etc etc

Granularity

- What is a data item (on which a lock is obtained)?
  - Most times, most systems: item is a 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

Explicit lock management

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

Implications

- For application programmer:
  - If transaction 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 transaction
  - Or rollback
Why it’s hard

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

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 txns 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 DBMSes 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 TPM onitor sofware) 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 snapshots
  - Snapshot isolation

Problem with serializability

- The performance reduction from isolation is high
  - Transactions are often blocked because they want to read data that another transaction has changed
- For some applications, the accuracy of the data they read is not crucial
  - e.g., overbooking a plane is acceptable
  - e.g., your banking decisions wouldn't 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
  - 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 transactions run at weaker levels!

Browse

- SET TRANSACTION 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 transaction to be read-only unless you say otherwise
  - A low-level transaction to read dirty data (from a transaction that will later abort)

Cursor stability

- SET TRANSACTION ISOLATION LEVEL
- READ COMMITTED
  - Set read locks but release them after the read has happened
    - e.g., when cursor advances onto another row during scan of the result of a multirow query
    - i.e., do not hold R-lock until commit or abort
  - S-x 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

13
**Repeatability read**

- `SET TRANSACTION 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.
  - 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 site is unavailable!
  - So allow replicas to be out-of-date.
  - Lazy propagation of updates.

**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.
- In effect, 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.

**Checks for conflict**

- Read of an item does not give current value.
- Instead, use the recovery log to find value that had been 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.
  - 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

**Problem s 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!

**Example - Skew Write**

Initial state of InStore, Product, Order

<table>
<thead>
<tr>
<th>p1</th>
<th>s1</th>
<th>p2</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>10</td>
<td>35</td>
<td>25</td>
</tr>
</tbody>
</table>

Update row 1: 30->4
Update row 2: 35->10

// No need to insert in Order

Find sum: 71

Note: sum uses old value of row 1 and Product, and self-changed value of row 2

Integrity constraint is false:
Sum is 46

Final state of InStore, Product, Order

<table>
<thead>
<tr>
<th>p1</th>
<th>s1</th>
<th>p2</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

**Implications**

- For the application programmer
  - Think carefully about your program's behavior if reads are inaccurate
  - Try possible workarounds if using 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!

**Skew Writes**

- SI breaks serializability when two modify different items, each based on a previous state of the item the other modified
- It is fairly rare in practice
- Eg the TPC-C benchmark runs correctly under SI when transactions 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

**To learn more**

- CSEP 545 Transaction Processing
  - Taught by Prof Phil Bernstein (Microsoft & UW adjunct)
  - Author of one of the best books on the subject (and inventor of some of the important ideas!)