EECS 262a
Advanced Topics in Computer Systems
Lecture 6

Transactions and Isolation Levels
September 14th, 2021

John Kubiatowicz

Based on slides by Alan Fekete, Uwe Roehm and Michael Cahill (University of Sydney),

updated by John Kubiatowicz and Anthony D. Joseph
Electrical Engineering and Computer Sciences
University of California, Berkeley
http://www.eecs.berkeley.edu/~kubitron/cs262a

Today's Papers

- Granularity of Locks and Degrees of Consistency in a Shared Database (2-up version)
  J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger. Appears In IFIP Working Conference on Modeling of Data Base Management Systems. 1975
- Principles of Transaction-Oriented Database Recovery, Theo Haerder and Andreas Reuter. Appears in *Journal of the ACM Computing Surveys* (CSUR), Vol 15, No 4 (Dec. 1983), pp287-317

- Thoughts?

Overview

- Transactions
  - ACID properties
  - Isolation Examples and counter-examples
- Classic 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):

```
Begin_Transaction()
Do a bunch of things - read and write data
End_Transaction()
```

  - Until End_Transaction() completes, transaction may be aborted and effects will be nullified
  - Transaction said to "Commit" after End_Transaction()
  - Once it has happened, it isn't forgotten (Durability)
- Commerce examples
  - Transfer money between accounts
  - Purchase a group of products
- Student record system
  - Register for a class (either waitlist or allocated)

ACID Semantics

- Full Transactional support includes:
  - Atomicity: It must be an all-or-nothing commitment
  - Consistency: After commit, a transaction will preserve the consistency of the data base
    - » Example: for banking app, net amount of money constant, even after moving money from account to account
  - Isolation: Events within a transaction hidden from other transactions
    - » Writes from uncommitted transactions invisible to other transactions
  - Durability: Once a transaction has been completed and committed its results, results will survive even system crashes
- Much of focus is around techniques for achieving these properties
  - Use of Log to permit transactions to abort/be durable

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

Threats to data integrity

- Need for application rollback
- System crash
- Concurrent activity
  - Today's lecture is about this

Provided by the log

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

Serializability

- To make isolation precise, we say that an execution is serializable when:
  - There exists some serial (i.e. 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 only ask that *some* serial order produces the given state
  - Other serial orders may give different final states
- Comparison with Sequential Consistency?
  - Close but not the same
  - Sequential consistency is: "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Model an execution as a sequence of operations on data items
    - » eg r1[x] w1[x] r2[y] r2[x] c1 c2
  - Serializability of an execution can be defined by equivalence to a rearranged sequence ("view serializability")
  - Treat the set of all serializable executions as an object of interest (called SR)
  - Thm: SR is in NP-Hard, 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!

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
- Classic Implementation Techniques
  - Locking
  - Lock Manager
  - Granularity of locks
- Weak isolation issues

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

Lock modes

- Locks can be for writing (X), reading (S)
  - Refinements have extra modes
- Standard conflict rules: two X locks on the same data item conflict, so do one X and one S lock on the same data
  - However, two S locks do not conflict
- Compatability Table:
  - X=exclusive
  - S=shared

Held/Requested	X	S
X	Block	Block
S	Block	OK

Strict two-phase locking

- Two phases:
  - Locks are being obtained (while txn runs)
  - Locks are released (when txn finished)
- 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) ← **NB. This is different from when locks are released in O/S or threaded code**

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*

Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which data, in which modes
- It does not allow 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 manager API

- Access mainly based on item's unique name
  - eg tupleid, or primary key, for records
- Lock(name, txn, mode)
  - Block until lock is available
- RemoveTxn(txn)
- Unlock(name, txn)
- LockUpgrade(name, txn, newmode)
- ConditionalLock(name, txn, mode)
  - Returns error immediately if unsuccessful

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 (especially in early 1970s): item is a page (with several tuples)
  - Sometimes: item is a whole table

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

Multigranular locking

- Care needed to manage conflicts properly among items of varying granularity
  - Note: conflicts only detectable among locks on a given itemname
- System gets "intention" mode locks on larger granules before getting actual S/X locks on smaller granules
  - Conflict rules arranged so that activities that do not commute must get conflicting locks on some item

Lock Mode Conflicts

Held\Request	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	Block
IX	Yes	Yes	Block	Block	Block
S	Yes	Block	Yes	Block	Block
SIX	Yes	Block	Block	Block	Block
X	Block	Block	Block	Block	Block

Other modes also used for special purposes, like select-for-later-update, gaplocks for phantom prevention

Lock manager may allow higher level to introduce its own conflict table

Lock manager internals

- Hash table, keyed by hash of item name
  - Each item has a mode and holder (set)
  - Wait queue of requests
  - All requests and locks in linked list from transaction information
  - Transaction table
    - » To allow thread rescheduling when blocking is finished
  - Deadlock detection
    - » Either cycle in waits-for graph, or just timeouts

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

Overview

- Transactions
- Classic Implementation Techniques
- Weak isolation issues
  - Including 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
- Recovery managers (e.g. Aries, next time) essential to enforcing A and D
  - Roll back values from uncommitted transactions
  - Restore values not yet committed to database

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 (Gray'75 called this "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 S-locks at all
    - » Of course, still set X-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)

Read Committed

- SET TRANACTION ISOLATION LEVEL READ COMMMITTED **Most common in practice!**
  - Set S-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 S-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
  - Gray'75 Called this "degree 2"

Repeatable read

- SET TRANACTION ISOLATION LEVEL REPEATABLE READ
  - Set S-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

Snapshot Isolation (SI)

- The transaction sees a "snapshot" of the database, at an earlier time
  - Intuition: this should be consistent, if the database was consistent before
- Read of an item may not give current value
- Instead, use old versions (kept with timestamps) 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

Snapshot Isolation (SI)

- A multiversion concurrency control mechanism was described in SIGMOD '95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil
  - Does not guarantee serializable execution!
- Supplied by Oracle DB, and PostgreSQL (before rel 9.1), for "Isolation Level Serializable"
- Available in Microsoft SQL Server 2005 as "Isolation Level Snapshot"

First committer wins (FCW)

- T will not be allowed to commit a modification to an item if any other transaction has committed a changed value for that item since T's start (snapshot)
- T must hold write locks on modified items at time of commit, to install them.
  - In practice, commit-duration write locks may be set when writes execute.
  - These simplify detection of conflicting modifications when T tries to write the item, instead of waiting till T tries to commit.

Benefits of SI

- Reading is never blocked, and reads don't block writes
- Avoids common anomalies
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)
- Matches common understanding of isolation: concurrent transactions are not aware of one another's changes

Is every execution serializable?

- For any set of txns, if they all run with Two Phase Locking, then every interleaved execution is serializable
- For some sets of txns, if they all run with SI, then every execution is serializable
  - Eg the txns making up TPC-C
- For some sets of txns, if they all run with SI, there can be non-serializable executions
  - Undeclared integrity constraints can be violated
  - We will revisit this on another day!

Is this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Second Paper: A survey of Recovery Techniques!

- The second paper is really a summary paper
  - Ideas behind transactions were stabilizing at this time (1983)
  - Gray, Berstein, Goodman, Codd, ….
- Point of reading this paper is to really start you thinking about recovery options
  - Preparation for ARIES paper (next time!)

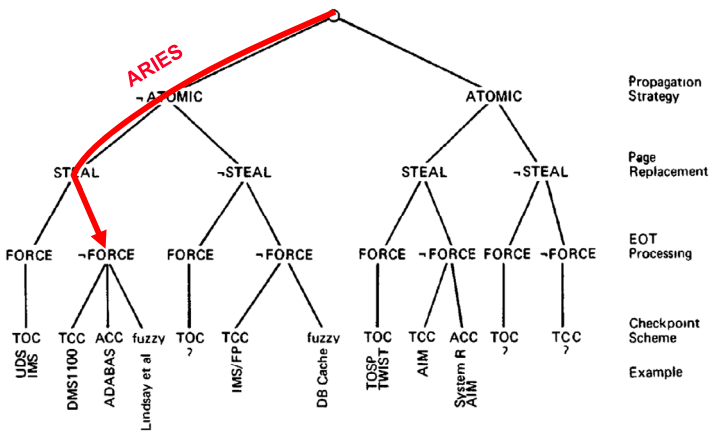
Recovery actions: The Log

- Log entries:
  - REDO: (forward) enough information to perform update to the data
  - UNDO: (backward) enough information to move backward\
  - BEGIN: Start of transaction
  - COMMIT: End of transaction
  - ABORT: Transaction was aborted
- Once transaction committed in log, transaction durable
  - Updates may not be reflected in permanent state
  - If crash happens before COMMIT is placed into log, should be as if nothing ever happened: may need to undo state
- Checkpoints:
  - Flush log out to well defined point/discard old log entries
  - Good point to recover from
- Ways of updating permanent state
  - Update in place: use log to undo state if necessary
  - Shadow pages: keep old state around and update to new pages
  - Other ideas: keep different views of DB, “Old and new” copies, etc

Many things in this paper:

- Lots of discussion of transaction implementation techniques (and buffer management)
  - ATOMIC, STEAL, FORCE, …
- Lots of discussion of logging techniques including how to deal with archive
- ARIES paper for next time will show “BEST IN CLASS” combination of features

Full Classification of Data Recovery Concepts



Was this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?