The University of Sydney

Topics in Database Isolation
IITB, January 2006
Lecture 1: Isolation Levels

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

Road Map

- Lecture 1: Isolation levels
  - Transaction Concept
  - ACID properties
  - Examples and counter-examples
  - Serializability Theory
  - Two phase locking and variants
  - Other SQL isolation levels
  - Snapshot Isolation
- Lecture 2: Safe Use of Low Isolation
- Lecture 3: Replication Management

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

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
  - Furthermore, once an outcome has been reached, it doesn't change

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 storeID = :store AND 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 of this product 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
  - Especially due to loss of DBMS buffers
  - Data on disk may be stale (and inconsistently so)
- Concurrent activity

- The system has mechanisms to handle these
  - Logging deals with rollback and crash recovery
    - Remember old value, restore it when needed
  - This talk is about concurrency issues

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 concurrency 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

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

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

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

60 returned p1's have vanished from system; total is still 115

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

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

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

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

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 application programs in which each app follows same logic and in which the overall final state is the same
  - 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

Execution is like serial MakeSale; ClearOut

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

Final state of InStore, Product, Order

Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Model an execution as a sequence of operations on data items
    - eg $r_1[x]\ w_1[x]\ r_2[y]\ r_2[x]\ c_1\ c_2$
  - 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

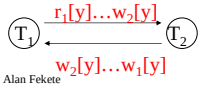
Conflict serializability

- 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!

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

- Items: Product(p1) as x, Instore(p1,s1) as y
- Execution is
  - $r_1[y]\ r_1[x]\ w_1[x]\ r_2[y]\ w_2[y]\ w_1[y]\ c_1\ c_2$
- Precedes Graph



DBMS support for transactions

- DBMS acts whenever application wants to read or write data
  - Or when explicitly told to act
- System's actions are to request/release "locks"
- This may cause temporary blocking, if locks are not available
  - There is also overhead while acting
- Strict two-phase locking ensures serializable transactions

Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which items, 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 (X), reading (S) or other modes
- Standard conflict rules: two X locks on the same item conflict, so do one X and one S lock on the same data
  - However, two S locks do not conflict
- Thus X=exclusive, S=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 ← NB. This is different from when locks are released in O/S or threaded code
  - 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 *every* 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 X-locks InStore. row 1
- update row 2: 70->5
- //t2 X-locks Instore.row2
- try find sum:// blocked
- // as S-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

Integrity constraint is valid

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

Example – No Lost update

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

Outcome is same as serial ClearOut; AcceptReturn

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

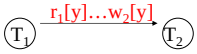
p1	etc	65
p2	etc	55
etc	etc	etc

Final state of InStore, Product

Example – No Lost update

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

- Product(p1) as x, Instore(p1,s1) as y
- $r_1[y]\ r_1[x]\ w_1[x]\ w_1[y]\ r_2[y]\ w_2[y]\ c_1\ c_2$



Granularity

- What is an 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 X/S locks on smaller granules
  - Conflict rules cover intention modes as well as X and S

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 tuple grain locks until there are too many of them; then it replaces them with page or table 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

Big Picture: 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 (serializable)
  - 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: Impact

- If programmer writes applications so each txn is consistent
- And DBMS uses logging and strict two-phase locking for every transaction
  - It 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!

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 S-locks till txn commits/aborts
    - Of course, still keep commit-duration X-locks
- Also called "level 1 isolation"

Impact of Read Committed

- Data seen by txn is never 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
- But performance is often much better
  - Bober and Carey (ICDE'92) simulation study shows approx 3 times higher throughput than for 2PL

Repeatable read

- SET TRANACTION ISOLATION LEVEL REPEATABLE READ
  - Set share and exclusive locks on data items, and hold them till txn finished, but release share 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

- Most DBMS vendors use variants of the standard locking algorithms
- However, recently a new "multiversion" concurrency control approach has become popular
  - Based on allowing readers to use old versions kept even after writer has changed an item
  - Note: this generalizes "MV2PL" described in textbooks by allowing reads of old versions in txns which do updates

Snapshot Isolation

- A multiversion concurrency control mechanism which was described in SIGMOD '95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil
- Used in Oracle, PostgreSQL for "Isolation Level Serializable"
  - But does not guarantee serializable execution as defined in standard transaction management theory
- Available in Microsoft SQL Server 2005 as "Isolation Level Snapshot"
  - Only available to a txn provided the database has had snapshots enabled

Snapshot Isolation (SI)

- Read of an item does 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
- 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 ww-conflict

- If a Snapshot transaction T has modified an item, T will not be allowed to commit if any other transaction has committed and *installed a changed value* for that item, between T's start (snapshot) and T's commit
  - "First committer wins"
  - Similar to optimistic validation-based cc, but only write-sets are checked
- T must hold X-lock on modified items at time of commit, to install them. In practice, commit-duration X-locks may be set when write executes. These help to allow conflicting modifications to be detected (and T aborted) when T tries to write the item, instead of waiting till T tries to commit.

Benefits of SI

- Reading is *never* blocked, and also doesn't block other txns activities
  - Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)

Problems with SI

- SI does not always give serializable executions
  - (despite Oracle etc using it for "ISOLATION LEVEL SERIALIZABLE)
  - Serializable: among two concurrent txns, one sees the effects of the other; versus SI: neither sees the effects of the other
- 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)
- Update row 1: 30->4
- MakeSale(p1,s2,25)
- 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

Integrity constraint is false: Sum is 46

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

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

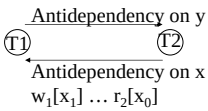
Multiversion Serializability Theory

- From Y. Raz in RIDE'93
  - Suitable for multiversion histories
  - Use subscript on item to indicate writer txn of that version
  - Eg $r_1[x_3]$ means T1 reads version of x produced by T3
- WW-conflict from T1 to T2
  - T1 writes a version of x, T2 writes a later version of x
    - In our case, succession (version order) defined by commit times of writer txns
- WR-conflict from T1 to T2
  - T1 writes a version of x, T2 reads this version of x (or a later version of x)
- RW-conflict from T1 to T2 (Adya et al ICDE'00 called this "antidependency")
  - T1 reads a version of x, T2 writes a later version of x
- Theorem: *Serializability of a given execution is proved by acyclic conflict graph*

Skew Writes

- Previous example
  - Item x: Instore(p1,s1)
  - Item y:Instore(p1,s2)
  - Item z:Product(p1)
- $r_1[x_0]$ $w_1[x_1]$ $r_2[y_0]$ $w_2[y_2]$ $r_2[x_0]$ $r_2[y_2]$ $r_2[z_0]$ $r_1[x_1]$ $r_1[y_0]$ $r_1[z_0]$ c_1 c_2

Conflict graph for this execution



Further Reading

- Transaction Theory: “Transactional Information Systems” by G. Weikum and G. Vossen
- Transaction implementation details: “Transaction Processing” by J. Gray and A. Reuter