

The University of Sydney
Transactions: Fekete

Transactions

Alan Fekete
(University of Sydney)
fekete@it.usyd.edu.au
April 2009

Transactions: Fekete

Road Map

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

2

Transactions: Fekete

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**)
 - 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)

3

Transactions: Fekete

Coding a transaction

- Typically a computer-based system doing OLTP has a collection of *application programs*
- Each program represents business logic of an activity, manipulating data in the DBMS
 - Application might be run separately, written in a special language extension with embedded SQL converted by preprocessor
 - Application might be run separately, written in a typical programming language making DBMS calls through a Call Level Interface (SQL string as argument)
 - Application code may be inside the DBMS as a stored procedure written in a dbms-specific language

4

Transactions: Fekete

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)

5

Transactions: Fekete

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

6

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”

7

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

8

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

9

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

10

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

11

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

12

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

13

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

14

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

15

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

16

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

17

Example - Constraints

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

18

Transactions: Fekete

Example - Constraints

- Data values
 - Instore.qty ≥ 0
 - Order.rcvd \leq 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

19

Transactions: Fekete

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

20

Transactions: Fekete

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

21

Transactions: Fekete

Example - Restock

- Input validation
 - Valid product, store, qty
 - Amount of product in warehouse \geq 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

22

Transactions: Fekete

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)

23

Transactions: Fekete

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?

24

Threats to data integrity

- Need for application rollback
- System crash
- Concurrent activity
- The system has mechanisms to handle these

25

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

26

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!

27

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

28

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)

29

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

30

Transactions: Fekete

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
- Phantom
 - One task T sees a row among those used in one query but not in another

31

Transactions: Fekete

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 32

Transactions: Fekete

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 33

Transactions: Fekete

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 34

Transactions: Fekete

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

35

Transactions: Fekete

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

Initial state of InStore, Product, Order

Order: empty

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 36

Transactions: Fekete

Serializability Theory

- There is a beautiful mathematical theory, based on complexity theory
 - 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-Complete, 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

37

Transactions: Fekete

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 T_i to T_j when T_i accesses x , then later T_j 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!

38

Transactions: Fekete

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

39

Transactions: Fekete

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

40

Transactions: Fekete

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!

41

Transactions: Fekete

Road Map

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

42

Main implementation techniques

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

43

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

44

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

45

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

46

Buffer management

- Each page has place for LSN of most recent change to that page
- When a page is held in buffer, DBMS remembers first LSN that modified the page
- 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; also write-ahead rule links flush of data page to previous flush of log); forced flush is expensive!

47

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"

48

Restart

- After a crash, the goal is to get to a state of the database which has the effects of those transactions that committed before the crash
 - it does not show effects of transactions that aborted or were active at the time of the crash
- To reach this state, 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

49

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, use information about the checkpoint to limit repeating history so it examines a suffix of the log
 - Tradeoff: aggressive checkpointing activity slows normal processing but reduces restart time

50

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

51

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
- These issues, and many others, are solved in ARIES paper by C. Mohan et al (ACM TODS in 1992)

52

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!

53

Main implementation techniques

- Logging
- **Locking**
 - Lock manager
 - Lock modes
 - Granularity
 - Scans and searches
 - User control
- Distributed Commit

54

Transactions: Fekete

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 correct lock is requested before every access to the data

55

Transactions: Fekete

Lock modes

- Locks can be for writing (X), reading (S) or other 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
- Thus X=exclusive, S=shared

56

Transactions: Fekete

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

57

Transactions: Fekete

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)

58

Transactions: Fekete

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 (and causing long delays till deadlock is detected and fixed)

59

Transactions: Fekete

Proof sketch

- Suppose all txns do strict 2PL
- If T_i has an edge to T_j in the precedes graph
 - That is, T_i accesses x before T_j has conflicting access to x
 - T_i has lock at time of its access, T_j has lock at time of its access
 - Since locks conflict, T_i must release its lock before T_j 's access to x
 - T_i completes before T_j accesses x
 - T_i completes before T_j 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*

60

Transactions: Fekete

Example – No Dirty data

- AcceptReturn(p1,s1,50) MakeSale(p1,s2,65)
- //t1 X-locks InStore.row 1
- Update row 1: 25 -> 75
- //t2 X-locks InStore.row2
- update row 2: 70->5
- 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

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 61

Integrity constraint is valid

Transactions: Fekete

Example – No Lost update

- ClearOut(p1,s1) AcceptReturn(p1,s1,60)
- //t1 S-lock InStore.row1
- Query InStore; qty is 25
- // t1 X-lock Product.row 1
- Add 25 to warehouseQty: 40->65
- try Update row 1
- // blocked
- // as X-lock on InStore.row1
- // can't be obtained
- //t1 upgrades to X-lock on InStore.row1
- Update row 1, setting it to 0
- COMMIT // release t1's locks
- // now get X-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 62

Outcome is same as serial
ClearOut; AcceptReturn

Transactions: Fekete

Example – No Lost update

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

• Items: Product(p1) as x, Instore(p1,s1) as y

• Execution is

$$- r_1[y] \ r_1[x] \ w_1[x] \ w_1[y]$$

$$- c_1 \ r_2[y] \ w_2[y] \ c_2$$

• Precedes Graph

(T1) $\xrightarrow{r_1[y]..w_2[y]}$ (T2)

No cycle: Serializable 63

Transactions: Fekete

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

64

Transactions: Fekete

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

65

Transactions: Fekete

Scans and Searches

- Serializability theory treats dbms as if all operations were read or write of a named item
- But SQL allows searching for items which meet a condition
 - Done at storage layer by look up in an index, or scan for “next record”
- Concurrency control is needed against concurrent transactions which insert, delete or modify records

66

Next Key Locks

- A transaction needs to lock the range that is examined in a scan or search
 - Often done by a lock on the key at the endpoint of the range
 - Even if the endpoint is found not to satisfy the condition, and thus is not returned
- Also, system gets lock on the “next key” after any record that is inserted
- There are several variations on details
 - eg are locks on range with usual modes, or special gap modes? Are locks on the key, or on the data record's tuple id, or on the index record?

67

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

68

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

69

Implementation mechanisms

- Logging
- Locking
- **Distributed Commit**

70

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

71

Why it's hard

- Imagine sending a message 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

72

Transactions: Fekete

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!)

73

Transactions: Fekete

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!

74

Transactions: Fekete

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

75

Transactions: Fekete

Fault-tolerant protocol

- The interchange of messages between the "coordinator" (part of the TP Monitor 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

76

Transactions: Fekete

Implications

- For application programmer
 - Avoid putting modifications to multiple databases in a single txn
 - Performance suffers a lot
 - X-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

77

Transactions: Fekete

Road Map

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

78

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

79

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

80

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!

81

Browse

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
 - Do not set S-locks at all
 - Of course, still set X-locks before updating data
 - In 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)

82

Read Committed

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED *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
- Also called "level 1 isolation"

83

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

84

Transactions: Fekete

Safe use of Read Committed

- Application semantics
 - Old data is acceptable
- Data semantics
 - Write-once data
 - No inter-item integrity constraints

85

Transactions: Fekete

Recently changed data

- Many applications can work well with slightly old data
 - Data used mainly as “hint”
 - Eg choose shipper based on expected delay
 - Social processes fix problems
 - Eg mail forwarding with obsolete address
 - Eg merchant honors old prices

86

Transactions: Fekete

Write-once data

- Data that never changes after commit of the transaction that inserted the record
 - Audit trail
 - Immutable attributes
 - Eg manufacturer of a product
- Note that we don’t need whole record to be constant, just those fields this particular application cares about

87

Transactions: Fekete

Unrelated items

- Application may deal with only one record
- Or it may deal with several records, but there are no integrity constraints between them
- Eg often each constraint concerns a single product, so an application that processes several products can safely release read locks on each product record before moving to the next one.

88

Transactions: Fekete

Repeatable read

- SET TRANSACTION 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

89

Transactions: Fekete

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

90

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

91

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

92

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 2005/8 as “Isolation Level Snapshot”
 - Only available to a txn provided the database has had snapshots enabled

93

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

94

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.

95

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)

96

Transactions: Fekete

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!

97

Transactions: Fekete

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

Example – Skew Write

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

- 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

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

p1	etc	32
p2	etc	44
etc	etc	etc

Order: empty₉₈

Final state of InStore, Product, Order

Integrity constraint is false:
Sum is 46

COMMIT

Transactions: Fekete

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

99

Transactions: Fekete

Multiversion Serializability Theory

- Several variants, we describe one 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 (sometimes called "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*

100

Transactions: Fekete

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

101

Transactions: Fekete

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!

102

Transactions: Fekete

Further Reading

- Main implementation techniques: “Architecture of a Database System” by J. Hellerstein, M. Stonebraker, and J. Hamilton, *Foundations and Trends in Databases* 1(2):141-259
- 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

103

Transactions: Fekete

Recent Transaction Research

- Properties of weak isolation
 - Declarative representation
 - Restricted cases where you still get integrity running with lower isolation level
- Extended transaction models
 - Suitable for web services, workflows
 - Across trust domains, so can't give up autonomy
- Weak isolation for internet-scale systems
 - Many replicas, which can't be kept consistent or always available
 - Various application needs (eg always write, but reads ¹⁰⁴ may be out-of-date)