



The University of Sydney
AC191842K

Database Transactions (Executive summary)

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

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, all at once) or it didn't happen at all
- Commerce examples
 - Transfer money between accounts
 - Purchase a group of products, and mark order as filled
- 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 parameterized *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
 - Can use program variables in queries
- A single SQL statement can't do it all!
 - Need to update several tables
 - Need control flow logic

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
- Even working interactively, all activity on dbms is in a txn
 - Each interactive SQL statement is separate txn

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
- Rollback can be due to:
 - Program request
 - Client process death
 - System-initiated (eg to deal with deadlock or crash)

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*
- Also, the decision is irrevocable; so once committed, the transaction can't revert to aborted
 - Changes are *durable*

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*
 - Perhaps, but not always, declared explicitly in the database schema
 - Eg foreign key constraint, primary key constraint

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 threats to data integrity

- Need for application rollback
 - Previous state must be re-installed
- System crash
 - Due to buffering, state after crash may be weird mix of pages: obsolete, over-fresh, accurate
- Concurrent activity
 - Data interference; avoided by isolation
- The system has mechanisms to handle these

Isolation

- To make precise what it means to have no data interference, 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

Serializability Theory

- There is a beautiful but useless mathematical theory, based on formal languages
- However, one can usually prove an algorithm gives serializable execution by using sufficient condition of conflict serializability: absence of cycles in “precedes” graph based on order of conflicting operations in different txns

ACID

- **A**tomic
 - State shows either all the effects of txn, or none of them
- **C**onsistent
 - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
 - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
 - Once a txn has committed, its effects remain in the database

Local to global reasoning

- 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!

Implementation techniques

- Logging
- Locking
- Two-phase commit
- Ideas, not details!
 - Subtle interactions between these, and with buffer management, dbms threads, etc

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
 - Often with old and new values explicitly
 - Sometimes logically (eg for B-tree splits)
- Also log changes due to rollback (“CLR”)
- Log also shows when txns start/commit/abort
- Log records identifies by LSN

Buffer management

- Each page has place for LSN of most recent change to that page
 - Also, 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): heavy performance hit!

Using the log

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

Restart

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

Optimizations

- Use LSNs recorded in each page of data, to avoid repeating changes already reflected in page
- Checkpoints: flush pages that have been in buffer too long (in background)
 - Record in log some information about this activity
 - NB disk does not have consistent set of pages!
 - During restart, only repeat history since start of last completed checkpoint

Complexities

- Multiple txns affecting the same page of disk
 - From “fine-grained locking” (see later)
- Partial rollback (to “savepoint”)
- 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
- Issues solved in ARIES papers by C. Mohan

Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which data, in which modes (S=R, X=W, U, etc)
 - Table controls which modes conflict
- 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
- Need for super-efficient implementation

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
- Locks are only released when txn completes (“strict two-phase locking”)

Complexities

- What is a data item (on which a lock is obtained)?
 - Most times, in most modern systems: item is one tuple in a table (“fine-grained lock”)
 - 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
- Also, special rules for locking index entries used to evaluate WHERE clause

Transactions across multiple DBMS

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

Why it's hard

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

Two-phase commit

NB unrelated to “two-phase locking”

- 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!)
- Commit decision can't happen till all DBMS involved are prepared
- DBMS surrenders its autonomy for a period
 - Vulnerable if coordinator is unavailable then!

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
 - But A and D are vital

Explicit isolation levels

- A transaction can be declared to have isolation properties that are less stringent than serializability
 - Implemented by releasing read locks early
 - 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!

Further Reading

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