9: Transactions

Last Modified:
10/8/2002 9:39:59 PM

Definition

- A transaction is a collection of instructions (or operations) that perform a single logical function.
- Customer buys a car
  - MerchantsInventory--
  - Customer Bank Account -=PRICE
  - Merchant Bank Account+=PRICE
  - CustomerHistory++
  - ….
- All of these things should happen indivisibly – all or nothing? Even in the presence of failures and multiple concurrently executing transactions!
- How do you make that happen when it is physically impossible to change all these things at the same time?

Commit/Abort

- Introduce concept of commit (or save) at the end of a transaction
- Until commit, all the individual operations that make up the transaction are pending
- At any point before the transaction is committed, it might also be aborted
- If a transaction is aborted, the system will undo or rollback the effects of any individual operations which have completed

Database Systems

- Manage transactions (much like OSes manage processes)
- Ensure the correct synchronization and the saving of modified data on transaction commit
- Databases and OSes have a lot in common!
- Databases get a better roadmap
  - SQL queries provide up front map of transactions data access intentions
  - General processes change pattern based on user input and are not as structured in their data access specifications
  - Some OSes provide APIs for programs to declare their intentions

ACID properties of Transactions

- (A)tomicity
  - Happen as a unit – all of nothing
- (C)onsistency
  - Integrity constraints on data are maintained
- (I)solation
  - Other transactions cannot see or interfere with the intermediate stages of a transaction
- (D)urability
  - Committed changes are reflected in the data permanently even in the face of failures in the system
- Atomicity, consistency and isolation are all the result of synchronization among transactions like the synchronization we have been studying between processes

Durability?

- How can we guarantee that committed changes are remembered even in the face of failures?
- Remembering = saving the data to some kind of storage device

Types of Storage

- Volatile Storage
  - DRAM memory loses its contents when the power is removed
- Non-Volatile Storage
  - Hard disks, floppy disks, CDs, tape drives are all examples of storage that does not lose its contents when power is removed
- Stable Storage
  - Still non-volatile storage can lose its contents (magnets, microwave ovens, sledge hammers,..)
  - "Stable storage" implies that the data has been backed up to multiple locations such that it is never lost

So what does this mean?

- Processes that run on in a computer system write the data they compute into registers, then into caches, then into DRAM
  - These are all volatile! (but they are also fast)
- To survive most common system crashes, data must be written from DRAM onto disk
  - This in non-volatile but much slower than DRAM
- To survive "all" crashes, the data must be duplicated to an off-site server or written to tape or ….. (how paranoid are you/how important is your data?)

ACID?

- So how are we going to guarantee that transactions fulfill all the ACID properties
  - Synchronize data access among multiple transactions
  - Make sure that before commit, all the changes are saved to at least non-volatile storage
  - Make sure that before commit we are able to undo any intermediate changes if an abort is requested
- How?

Log-Based Recovery

- While running a transaction, do not make changes to the real data; instead make notes in a log about what *would* change
- Anytime before commit can just purge the records from the log
- At commit time, write a "commit" record in the log so that even if you crash immediately after that you will find these notes on non-volatile storage after rebooting
- Only after commit, process these notes into real changes to the data

Log records

- Transaction Name or Id
  - Is this part of a commit or an abort?
- Data Item Name
  - What will change?
- Old Value
- New Value

Recovery After Crash

- Read log
- If see operations for a transaction but not transaction commit, then undo those operations
- If see the commit, then redo the transaction to make sure that its affects are durable
- 2 phases – look for all committed then go back and look for all their intermediate operations

Making recovery faster

- Reading the whole log can be quite time consuming
  - If log is long then transactions at beginning are likely to already have been incorporated.
- Therefore, the system can periodically write outs its entire state and then discard the log to that point
- This is called a checkpoint
- In the case of recovery, the system just needs to read in the last checkpoint and process the log that came after it

Synchronization

- Just like the execution of our critical sections
- The final state of multiple transactions running must the same as if they ran one after another in isolation
  - We could just have all transactions share a lock such that only one runs at a time
  - Does that sound like a good idea for some huge transaction processing system (like airline reservations say?)
- We would like as much concurrency among transactions as possible

Serializability

- Serial execution of transaction A and B
  - Op 1 in transaction A
  - Op 2 in transaction A
  - ....
  - Op N in transaction A
  - Op 1 in transaction B
  - Op 2 in transaction B
  - …
  - Op N in transaction B
- All of A before any of B
- Note: Does not apply outcome of A then B is same and B then A!

Serializability

- Certainly strictly serial access provides atomicity, consistency and isolation
  - One lock and each transaction must hold it for the whole time
- Relax this by allowing the overlap of non-conflicting operations
- Also allow possibly conflicting operations to proceed in parallel and then abort one only if detect conflict

Timestamp-Based Protocols

- Method for selecting the order among conflicting transactions
- Associate with each transaction a number which is the timestamp or clock value when the transaction begins executing
- Associate with each data item the largest timestamp of any transaction that wrote the item and another the largest timestamp of a transaction reading the item

Timestamp-Ordering

- If timestamp of transaction wanting to read data < write timestamp on the data then it would have needed to read a value already overwritten so abort the reading transaction
- If timestamp if transaction wanting to read data < read timestamp on the data then the last read would be invalid but it is commited so abort the writing transaction
- Ability to abort is crucial!

Outtakes

Is logging expensive?

- Yes and no
  - Yes because it requires two writes to nonvolatile storage (disk)
  - Not necessarily because each of these two writes can be done more efficiently than the original
    - Logging is sequential
    - Playing the log can be reordered for efficient disk access

Deadlock

- We'd also like to avoid deadlock among transactions
- Common solution here is breaking "hold and wait"
- Two phase locking approach
  - Generalization of getting all the locks you need at once then just release them as you no longer need them
  - Growing phase – transaction may obtain locks but not release any
    - Violates hold and wait?
  - Shrinking phase – transaction may release locks but not obtain any