Concurrency Control and Recovery in Databases

Outline

- Abstract model that databases try to provide
- Typical data structures used in the database
- Concurrency control and recovery process

Context

- Concurrency is needed for performance (multi-core, overlap I/O)
- Concurrency creates consistency problems ("concurrency control" problem)
- Failures happen! ("recovery" problem)

Programming Model

- Transaction: unit of program execution that accesses/ updates various data items
- It consists of all operations between:
  - BEGIN TRANSACTION
  - COMMIT or ABORT (end of the transaction)
- System R provides a few additional actions:
  - SAVE: intermediate results inside a transaction
  - UNDO: rollback to the previous SAVE point
  - READ_SAVE: read the contents of the last SAVE point

Transactions

- What are the pros and cons of using a transaction based model?
- System could fail during a transaction
  - What are the different kinds of failures?
  - Which are easy/hard?

Implementing Transactions

- Inconsistent executions due to concurrency:
  - ***Lost Update***: two tasks both modify the same data
  - ***Inconsistent Read***: one task sees some but not all of the changes made by another task
  - ***Dirty Read***: a task reads data updated by another task which will eventually abort

Formalizing Correctness

- Atomic: state shows either all the effects of a transaction or none of them
- Consistent: transaction moves only between states where integrity holds
- Isolated: effects of transactions is the same as transactions running one after another
- Durable: once a transaction has committed, its effects remain in the database

ACID: Notes

- Consistency: database satisfied integrity constraints
  - Account numbers are unique
  - Sum of debits and credits is zero
- Introduced as a requirement, but today we understand it as a consequence of:
  - correct programs
  - atomicity and isolation guarantees

Serializability

- Conflicting operations:
  - two updates to the same location
  - an update and an access to the same location
- Serializability check:
  - Order conflicting operations from different transactions
  - All ordering constraints between two transactions should go in the same direction (i.e., T1's operations happened before T2's operations or the other way around)
  - When do databases use the serializability check?

Locking

- Two approaches to concurrency control:
  - Use locking to ensure mutual access
  - Optimistic concurrency control: don't use lock and check for inconsistencies when the operation commits
- What are the tradeoffs between the two forms of concurrency control?

Locking Concepts

- Well-formed transactions:
  - Transaction holds lock (read or write lock) on the object when it performs the corresponding operation
  - Not sufficient for serializability
- Two-phase locking: transactions have two phases
  - Growing phase: in which transaction is acquiring locks
  - Shrinking phase: in which locks are released
- Need to deal with deadlocks using traditional schemes

Recovery

- Question:
  - What data structures are needed for recovery?
  - Feel free to propose other options than what is discussed in the paper/chapter

Data Structures

- Two kinds of storage: volatile (memory) and non-volatile (disk)
- Buffer pool: accessed or modified pages in memory
- Pages on disk
  - current version and possibly a shadow version
- Log of operations on disk (typically a write-ahead log)

Stable Storage

- STEAL: buffer manager allows the disk version to be updated even before the transaction is completed
- NO-STEAL: all updates made after the transaction is completed
- FORCE: all updates are reflected on disk before the transaction is allowed to commit
- NO-FORCE: transaction commits before updates are on disk

- Question:
  - Why do we want STEAL?
  - Why do we want NO-FORCE?

Logging

- UNDO: rollback updates on disk for uncommitted transactions
- REDO: make updates to disk for committed transactions
- Log is used to keep track of what to UNDO and REDO
- Log records contain a Log Sequence Number (LSN); data values (on disk) keep track of the LSN of update

Write Ahead Logging

- All log records pertaining to an updated page are written to disk before the page itself is modified on disk
- Transaction is not considered committed until all of its log records are on disk

Logging

- Two types of logging:
  - Physical: For every log entry, maintain the "before image" and "after image" of the updated data value
  - Logical: Keep track of what operation was performed (say increment of a value, insertion of a new tuple in a list, etc.)
- What are the tradeoffs between the two types of logging?

Data Structures

- Transaction Table: contains status information of active transactions
- Dirty pages table:
  - entries contain “recoveryLSN”: LSN of log record that made the page dirty
- Log records of a transaction:
  - contain prevLSN linking previous operations of the transaction
- Checkpoint records:
  - currently active transactions
  - dirty pages corresponding to these ongoing transactions

Recovery

- Three stages:
  - Analysis
  - REDO phase
  - UNDO phase

Analysis

- Determine the point to start the REDO pass
- Determine which pages could have been dirty at the time of the crash to avoid unnecessary I/O
- Determine which transactions had not committed

REDO

- Minimize disk I/Os
  - If affected page is not on the Dirty Page Table, then don't REDO
  - If affected page is in the Dirty Page Table, then if the recoveryLSN in the page table is greater than the LSN of the record being checked, then don't REDO
  - Check LSN on the page on the disk. If pageLSN is greater than or equal to the LSN, then don't REDO

UNDO

- Go back and unroll all uncommitted transactions
- Handle failures during recovery:
  - Maintain "Compensation Log Record" to keep track of what has been undone
  - Store this also in the log