Transaction Management

15-712: Software Systems

11 February 1999

Motivation

- Database operations are sequences of slow I/O ops interspersed by fast computation
- Want to overlap database operations to maximize CPU usage - concurrency
- Need to ensure that concurrent operations preserve program correctness

Transactions

- *Transaction*: a collection of actions in a recovery unit
- Desired properties - *ACID*:
  - Atomic
  - Consistent
  - Isolated
  - Durable

Executing Transactions

- Changes to data state are not globally visible until the transaction *commits*
- Changes are never visible if it *aborts*
- Start and end of execution delimited:
  - `BEGIN_TRANSACTION`
  - `COMMIT_TRANSACTION`
  - `ABORT_TRANSACTION`

Concurrent Transactions

- Overlap transactions for concurrency
- Multiple transactions may share data
- Data dependencies may cause errors:
  - Lost updates - write $\rightarrow$ write
  - Dirty reads - write $\rightarrow$ read
  - Unrepeatable reads - read $\rightarrow$ write
- Concurrency errors break consistency

Concurrency and Consistency

- Programmer must enforce consistency
- Simple way: execute transactions one at a time, in a *serial* order
  - Guarantees ACI properties
- Complex way: interleave executions, but ensure that it is equivalent to a serial order

Consistency and Serializability

- Assume: a serial execution is consistent
- Claim: an set of executions is consistent if it is equivalent to some serial execution
- 'Proof':
  - T < T' means T' has a data dependency on T
  - BEFORE(T) = {T' | T' < T}, AFTER(T) = {T' | T < T'}
  - If some T'' appears in BEFORE(T) and AFTER(T), then the execution is not equivalent to some serial execution
  - Execution is therefore not consistent

Example of Inconsistency

- T1: BEGIN
- T1: WRITE A
- T1: WRITE B
- T1: COMMIT
- T2: BEGIN
- T2: READ A
- T2: READ B
- T2: COMMIT

- T1: BEGIN
- T1: WRITE A
- T2: BEGIN
- T2: READ A (!!)
- T2: READ B
- T2: COMMIT
- T1: WRITE B
- T1: COMMIT

Serializability and Locks

- Use locks to avoid concurrency errors
- Basic locks: *shared* and *exclusive*
- Define *compatibility*:
  - Can obtain multiple shared locks on data
  - Can obtain an exclusive lock on data if no other locks are already held on it
- Can *upgrade* from shared to exclusive

Fixed Inconsistency

- `T1: BEGIN`
- `T1: X-LOCK A`
- `T1: WRITE A`
- `T2: BEGIN`
- `T2:` (try `SH-LOCK A`)
- `T1: X-LOCK B`
- `T1: WRITE B`
- `T1: COMMIT`
- `T2: SH-LOCK A`
- `T2: READ A`
- `T2: SH-LOCK B`
- `T2: READ B`
- `T2: COMMIT`

Lock Protocol for Consistency

- Claim: following protocol preserves serializability, and therefore consistency:
  - Obtain shared lock to read data
  - Obtain exclusive lock to write data
  - Hold all locks until committing
- Proof:
  - Shared lock prevents read $\rightarrow$ write errors
  - Exclusive lock prevents write $\rightarrow$ {read, write} errors
  - Hold ensures serializable execution

Lock Granularity

- Simple locking: lock entire record
  - Safe
  - Discards potential concurrency
- Complex locking: lock parts of a record
  - Admits greater concurrency
  - Need to show that new protocols guarantee serializability

Hierarchical Locks

Database

|

Index nodes

|

Record nodes

|

Field nodes

- Arrange database in a hierarchy of structures
- Locking a node implicitly locks the descendents
- Need to add *intention* locks to prevent conflicting locking of ancestors

Locks and Deadlock

- *Deadlock* occurs when two transactions attempt to obtain incompatible locks
- Example:
  - Transaction T1 gets shared lock on data D1
  - Transaction T2 gets shared lock on data D2
  - T1 tries to get exclusive lock on D2
  - T2 tries to get exclusive lock on D1

Deadlock Detection

- Construct *wait-for graph* of dependencies between transactions and locks
- Deadlock has occurred if graph has cycles
- When to check?
  - At each lock request
  - Periodically

Summary: ACID Revisited

- Adherence to the locking protocol yields serializable executions, thus isolation
- Isolation effectively yields correctness

- Commit / abort and recovery protocols yield atomicity
- Storage and backup system yield durability