Concurrency Control

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

Practice Quiz: Query Processing and Optimization

- Working with a neighbor:
  - Draw a precedence graph for the transactions shown in the table
  - Determine whether the transactions are conflict serializable

T_1	T_2	T_3
read A		
	read A	
	write A	
		read A
	read B	
		read B
		write B
		write C
write C		

Agenda

- Attendance Quiz
- Locking Protocols
- Other Serializability Approaches
- Other Issues

Motivation for Concurrent Processing

- Effective use of system resources
  - Do work on CPU while waiting for a disk access
  - Do multiple disk accesses on multiple disks in parallel
- Support multiple simultaneous database users/sessions
- Take advantage of idle time during interactive transactions
- Keep the database accessible during a long-running transaction

Requirements for Concurrency

- Need to ensure the following for concurrent transactions
  - Serializability – equivalent to a serial schedule
    - Maintains consistency
  - Recoverability – a transaction cannot commit until any transaction whose data it uses commits
- How are these requirements actually implemented?

Locking Protocols

Locks

- A *locking protocol* is a set of rules which ensure that any schedule developing over time is serializable
  - More pragmatic than testing for serializability since future transactions (usually) cannot be predicted
- New database primitives
  - *Lock* – exclude other transactions from accessing a certain data item
  - *Unlock* – releases a predefined lock on a data item
- Locks often persist until the end of a transaction
- Locks are implicit to database operations
  - No need to tell a database to lock an item; it knows when to do so
  - This lecture shows locks explicitly to help illustrate them

Granularity of Locks

- Database locking – the entire database is locked (create or drop database)
- File locking – all objects in a file become unusable by other transactions
  - Used for growing, shrinking, or reorganizing files
  - "Online" mode can cause this work to happen in the background and then be switched into place once it completes
- Database object locking – tables, indexes, etc.
  - Used when altering the object's structure (via DDL statement)
    - Adding a column to a table
    - Rebuilding an index
- Record (row) or field (column) locking – a single tuple or data item is locked during a transaction
- Block level locking – common because data is read and written in blocks
  - A transaction may lock not only the record it is using, but the other records on the block as well

Shared Locks

- Used when a transaction reads an item without changing it
- Other transactions may also obtain shared locks on the item
- Shared lock prevents the data item from being changed while the transaction(s) read it
- Example: read current account balance

  lock-s(balance)

  read(balance)

  unlock(balance)

  - If the transaction is reading balances on multiple accounts, it needs to obtain shared locks on each of them

Exclusive Locks

- Used when a transaction **writes** an item (also allows for **reading** the item)
- A transaction seeking an exclusive lock must wait until all other locks on the desired item are released
- No other transaction can obtain any kind of lock on an item while an exclusive lock is held on it
  - Exclusive lock remains in force until the transaction commits or rolls back
- Read-modify-write operation
  - Obtain an exclusive lock before reading the item OR
  - Obtain a shared lock for the read, and then upgrade to an exclusive lock before the write
- Example: post interest to account (without and with lock upgrading)

Without lock upgrading	With lock upgrading
lock-x(balance)	lock-s(balance)
read(balance)	read(balance)
write(balance)	upgrade(balance)
unlock(balance)	write(balance)
	unlock(balance)

Deadlock

- Problem that can arise with locking protocols between transactions
  - Transaction T_1 has a lock on resource R_1 and needs a lock on resource R_2 before it can unlock R_1
  - Transaction T_2 has a lock on resource R_2 and needs a lock on resource R_1 before it can unlock R_2
- Example: Transfer $50 from checking to savings while printing total of account balances

Transfer (T_1)	Balance Inquiry (T_2)
lock-x(checking balance) read(checking balance) calculate new balance = old – 50 write(checking balance)	
	lock-s(savings balance) read(savings balance) lock-s(checking balance) – *must wait*
lock-x(savings balance) – *must wait*	

Dealing with Deadlock

- Approaches
  - Deadlock Prevention – design a scheme that stops deadlock from ever occurring (not always possible)
  - Deadlock Avoidance – Delay any lock which could lead to deadlock (Requires some advance knowledge of how transactions will behave)
  - Deadlock Detection and Recovery – Allow deadlock, and when it occurs, rollback one of the transactions and restart it after the other proceeds past the point of deadlock

- Most DBMS's use deadlock detection and recovery
  - Databases usually have lots of small transactions, decreasing the probability of deadlock
    - Databases need to support rollback anyway
  - Not a good approach to deadlock at the OS level (high rollback cost)

Locking by Itself is not Enough

Transfer (T_1)	Balance Inquiry (T_2)
	lock-s(savings balance)
	read savings balance (S)
lock-x(checking balance)	
read checking balance (C)	
write checking balance (C-50)	
unlock(checking balance)	lock-s(checking balance)
	read savings balance (C-50)
	unlock(savings balance)
	unlock(checking balance)
lock-x(savings balance)	
read savings balance (S)	
write savings balance (S+50)	
unlock(savings balance)	

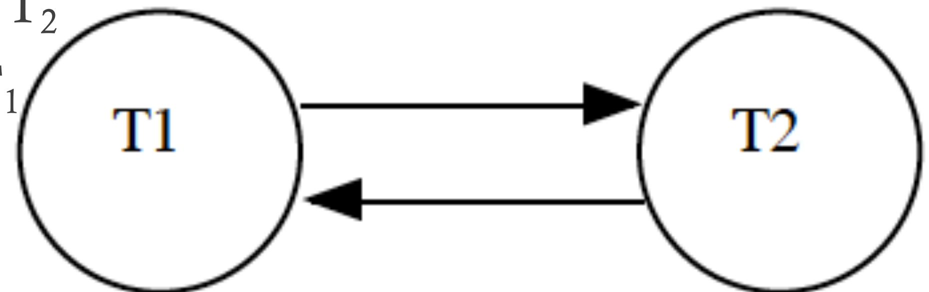
- Each transaction obtains appropriate locks
  - But there is still an error in the Inquiry transaction's balance total

Two-Phase Locking Protocol

- Governs the order in which transactions acquire and release locks
- Requires that a transaction must acquire all the locks it needs before releasing any of them
  - *Growth phase* – transaction acquires locks, but may not release any
    - Includes upgrading locks
  - *Shrinking phase* – transaction may release locks, but may not acquire any more
    - Includes downgrading locks (i.e. from exclusive to shared)

Two-Phase Locking and Transaction Serializability

- Two-phase locking can be used to ensure serializability
- Extension to precedence graph (used to test for conflict serializability)
- Directed edge for a *precedes relationship*
  - T_1 precedes T_2 ($T_1 \rightarrow T_2$) if in some schedule T_1 acquires a lock on some resource R before T_2 acquires an incompatible lock on R
  - If the precedence graph is acyclic, the schedule is serializable
- Example: transfer (T_1) and inquiry (T_2)
  - T_1 locks checking balance before T_2
  - T_2 locks savings balance before T_1
  - Cycle in graph, so not serializable



Two-Phase Locking and Transaction Recoverability

- Extensions to two-phase locking protocol
  - *Strict* two-phase protocol requires that all exclusive locks be held until a transaction commits
  - *Rigorous* two-phase protocol requires that all locks (shared or exclusive) be held until a transaction commits
- Both of these variants guarantee cascade-less recoverability, because no transaction can read data written by an uncommitted transaction
- Both variants are widely used along with some deadlock detection and recovery mechanism
  - Since two-phase locking can lead to deadlock

Other Serializability Approaches and Issues

Other Methods to Ensure Serializability

- Timestamps
- Validation
- Multiversion Schemes

Timestamps

- Each transaction is issued a unique serial number/clock reading when it starts
  - If an old transaction T_1 has time-stamp TS(T_1), a new transaction T_2 is assigned time-stamp TS(T_2) such that $TS(T_1) < TS(T_2)$
- Timestamps ensure that a transaction schedule is equivalent to a serial schedule
  - T_1 completes before T_2 because $TS(T_1) < TS(T_2)$
  - Stops reads or writes that would lead to a non-serializable schedule (like locking)
- Each data item Q maintains two timestamp values
  - W-timestamp(Q) – largest timestamp of any transaction that successfully wrote to Q
  - R-timestamp(Q) – largest timestamp of any transaction that successfully read Q
  - Conflicting read and write operations are executed in timestamp order
- Can have cascading rollbacks

Validation

- Allow transaction to read and write freely, but before it commits, ensure the outcome is serializable
  - Optimistic concurrency control – transaction fully executes "hoping" that validation goes well
- Allows higher levels of concurrency
  - Good if most transactions are read-only and do not interfere with each other

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
  - The old version(s) also retained
  - Use timestamps to label versions.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.
- Requires extra storage for versioned tuples and versioning data

Other Issues

- Deletes, Inserts, and Phantom Rows
- Weak Levels of Consistency
- Locking and Index Structures

Deletes and Inserts

- Inserts and deletes are like write operations (with regard to an entire row)
- Consider the following query:
  select count(*) from checked_out where borrower_id = 12345
  - What happens if a concurrent transaction does an insert or delete of a row with borrower_id = 12345?
    - If the operation is "ahead" of the select, it impacts the count
    - If the operation is "behind" the select, it does not impact the count
      - This *phantom row* is a problem.
- Solution: make doing an insert or delete a lockable operations
  - Insert/delete obtains an exclusive lock on this ability before executing
  - Count operation obtains a shared lock to prevent other rows from being inserted or deleted while it runs
  - Does not lock the whole table – other transactions can continue to run

Weak Levels of Consistency

- Ensuring serializable schedules takes overhead to either
  - Require transactions to wait for lock(s) to release before proceeding
  - Roll back transactions performing operations that would lead to a non-serializable schedule (and potentially restart them)

- Serializability enforcement can be relaxed if an approximate answer is close enough
  - Different levels of weakened serializability supported by SQL
    - **Serializable:** enforces full serializability
    - **Repeatable read:** allows only committed records to be read, and repeating a read within a single transaction should return the same value (Other transactions cannot change the value between successive reads)
      - However, phantom rows are still possible
        - T1 may see some records inserted by T2, but may not see others inserted by T2
    - **Read committed:** only committed records can be read, and repeating a read within a single transaction might return different values (if some other transaction changes the data item)
    - **Read uncommitted:** allows even uncommitted data to be read (dirty read)

PostgreSQL Isolation Levels

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Read committed is the default in PostgreSQL

https://www.postgresql.org/docs/13/transaction-iso.html

Locks and Index Structures

- What happens to indexes when the data they reference gets locked
  - A transaction looking up data via an index (e.g. read) needs shared locks on all index leaf nodes that it uses
  - A transaction doing inserts, updates, or deletes (e.g. write) needs exclusive locks on all leaf nodes affected by the operation
    - Also needs to update all pertinent indexes
- Indexes are accessed very often, so some index locking protocols do not require two phases
  - Accuracy is still required
  - Need for speed trumps serializability